Fachhochschule Köln
Cologne University of Applied Sciences

MASTERARBEIT

vorgelegt an der Fachhochschule Köln

Campus Gummersbach

im Studiengang Medieninformatik

Auswirkungen des Architekturstils REST auf die Anwendungsentwicklung

ausgearbeitet von
Thomas Heß
mail@thomashess.net

vorgelegt am
1. Juli 2011

erster Prüfer
Prof. Dr. Kristian Fischer

zweiter Prüfer
Prof. Dr. Mario Winter

Zusammenfassung

Durch die stärkere Verbreitung des Architekturstils REST kommt es zu Veränderungen in der Softwareentwicklung. Dabei wirft vor allem das Umfeld von Unternehmenssoftware einige Herausforderung auf, da dort bisher operationsorientierte Dienste eine dominierende Rolle einnehmen. Diese Arbeit untersucht, welche Auswirkungen die Verwendung von REST-Architekturen auf die Anwendungsentwicklung besitzt. Durch den Unternehmenskontext geht es hierbei vor allem um die Umsetzung von Integrationsszenarien und nicht um einen Vergleich, der eine bestimmte Technologie oder ein bestimmtes Konzept als überlegen darstellt.

Abstract

The growing popularity of the architectural style REST leads to changes in software development. Especially the domain of enterprise software offers multiple challenges, because by today operation oriented services are dominating this field. This thesis investigates, how RESTful architectures influence the development of software applications. The context of enterprise software leads to an emphasis on integration scenarios instead of a comparison, that aims for superiority of a certain technology or concept.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

BPEL Business Process Execution Language

BPMN Business Process Model and Notation

HATEOAS Hypermedia As The Engine Of Application State

HTTP Hypertext Transfer Protocol

IoC Inversion of Control

JAX Java API for XML

JAX-RS Java API for RESTful Web Services

JSON JavaScript Object Notation

MVC Model View Controller

POJO Plain Old Java Object

ReLL Resource Linking Language

REST Representational State Transfer

RESTful konform zu den REST-Prinzipien

RPC Remote Procedure Call

SOA Service Oriented Architecture

SOAP ehemals Simple Object Access Protocol

URI Uniform Resource Identifier

WADL Web Application Description Language

WSDL Web Service Description Language

WS* Technologie-Stack um SOAP herum

XML Extensible Markup Language

XPath XML Path

XSTL Extensible Stylesheet Language Transformations

1 Einleitung

1.1 Motivation

Innerhalb eines Unternehmens wird Software eingesetzt, um die dort vorhandenen Geschäftsprozesse zu unterstützen. Aus der Softwarekriese haben Unternehmen und Entwickler gelernt, dass Software nicht monolithische Konstrukte sein sollten. Stattdessen ist es sinnvoller, entkoppelte Komponenten zu entwickeln, die bei Bedarf ausgetauscht, erweitert oder angepasst werden können, ohne dass fremde Komponenten durch diese Änderungen negativ beeinflusst werden. Dies ist mit ein Grund dafür, weshalb verteilte Anwendungen inzwischen so stark verbreitet sind.

In den 90er Jahren wurde von Unternehmen die Geschäftsprozessmodellierung entdeckt (siehe [46]). Man erhofft sich davon, dass durch solche Modelle eine Grundlage geschaffen wird, um die Prozesse der Unternehmen zu optimieren. Seitdem die Modellierung von Geschäftsprozessen das Interesse von Unternehmen und Forschung geweckt hat, sind unterschiedliche Sprachen und Notationen entstanden, um die Prozesse zu beschreiben. Hierbei kommen jeweils unterschiedliche Ausgangsintentionen zum Tragen, was mit der Modellierung konkret bezweckt werden soll. Im Rahmen dieser Masterarbeit ist vor allem die Automatisierung von Prozessen durch die Komposition von Diensten von Bedeutung. In einem solchen Szenario ist es stets die Herausforderung, die Fachseite der Anwendungsdomäne und die Softwareentwickler zusammenzuführen und dabei die Verantwortlichkeiten sauber voneinander zu trennen.

Eine weitere Innovation der 90er Jahre war die Entstehung des Web. Auf Grund dessen starken Wachstums erkannten Unternehmen, dass das Web für mehr als nur eine zusätzliche Visitenkarte genutzt werden kann. Das Web bietet einen großen Markt von Kunden und Kontakt zu möglichen Geschäftspartnern. Diese Beziehungen sind im Zusammenhang mit der Modellierung von Geschäftsprozessen interessant, da sich das Web durch die Prozessketten durchzieht bzw. die Prozessketten durch Kommunikation über Webtechnologien optimiert werden können.

Mit SOAP wurde ein Protokoll entwickelt, dass sich für dieses Szenario der Kommunikation zwischen Unternehmen über das Web eignet. Die dem Web zugrunde gelegten Technologien bieten eine Möglichkeit, die Heterogenität der kommunizierenden Systeme zu überwinden, wie es bereits durch Webbrowser gezeigt wurde. So kommt es, dass SOAP auch ein beliebtes Protokoll ist, um die organisationsinternen Systeme miteinander zu verbinden.

Roy T. Fielding sieht im Web jedoch mehr Nutzen als das Überwinden von Heterogenität. Er zeigt auf, wie das Web den Zustand von Anwendungen über Hypermedia steuert oder Informationen über ein einheitliches Schema adressiert werden können. Diese und noch weitere Aspekte werden später noch ausführlicher in dieser Arbeit betrachtet. Für Fielding bietet das Web die Möglichkeit, entkoppelte Systemkomponenten zu entwickeln, genau so, wie auch Webseiten voneinander entkoppelt weiterentwickelt werden können und einan-

der trotzdem referenzieren können. Dieser Architekturstil mit dem Namen REST wurde in der Dissertation von Fielding im Jahr 2000 vorgestellt. Inzwischen ist dieser Architekturstil auch auf Interesse in der Praxis gestoßen, so dass führende Internetunternehmen für ihre Dienste RESTful APIs anbieten und auf SOAP basierte Schnittstellen nicht mehr weiterpflegen. Ein prominenteres Beispiel hierzu wäre die „Google Search API“. Für die Zukunft ist davon auszugehen, dass REST mehr an Bedeutung gewinnen und somit auch die Anwendungsentwicklung verändern wird.

1.2 Problemstellung

Diese Masterarbeit widmet sich dem Szenario, dass innerhalb von Unternehmen in die existierende Dienstlandschaft neue Dienste, die nach dem REST-Architekturstil arbeiten, integriert werden sollen. Dabei ist die Besonderheit, dass die bisherigen Dienste und Dienstkompositionen mittels SOAP, WSDL und BPEL implementiert sind, da es sich hierbei um ein in der Praxis stark verbreitetes Tripel handelt.

Dabei wirft REST mehrere Probleme auf, die betrachtet werden sollen:

1. Der *Wegfall einer maschinenlesbaren Schnittstellenbeschreibung* führt dazu, dass Werkzeuge zum Entwurf von Prozessen bzw. Prozessausführungsumgebungen, wie sie aus dem BPEL-Umfeld bekannt sind, keine Grundlage haben, um den funktionalen Umfang eines RESTful Dienstes zu bestimmen.

2. Die *Semantik der Dienste* wird jeweils von der Fachseite bzw. den Softwareentwicklern in einer eigenen Sprache formuliert, so dass im Dialog eine Übersetzung gefunden werden muss. Diese Problematik ist seit Beginn des IT-Einsatzes in Unternehmen präsent. Sie muss nun jedoch neu betrachtet werden, da diese Bedeutungen üblicherweise bei der Definition der Schnittstellenbeschreibung geklärt werden, welche bei REST jedoch nicht mehr bzw. in einer anderen Form existiert.

3. REST kann sein volles Potential erst dann entfalten, wenn der *Hypermedia-Constraint* sowohl von den Diensten als auch den Clients beachtet wird. Hierbei stellen jedoch besonders die Client-Implementierungen eine größere Herausforderung dar, da diese ihre Interaktionen mit dem Dienst nach den Zuständen der beteiligten Ressourcen richten müssen.

4. Die *Integration von SOAP-getrieben mit RESTful arbeitenden Diensten* innerhalb einer Prozessausführungsumgebung verlangt eine entsprechende Prozessengine und Entwicklungswerkzeuge. Die Kombination unterschiedlicher Architekturstile ist notwendig, da kein Unternehmen seine gesamte IT in einem Entwicklungsschritt einem Architekturwandel unterwerfen würde. Mögliche Gründe hierfür sind Entwicklungskosten, fehlendes Fachwissen, unerwartete Komplikationen und noch nicht abgeschriebene, bereits vorhandene Systeme.

Trotz der benannten Probleme bietet der Einsatz von REST einige Chancen: Zum einen legt REST kein Repräsentationsformat fest und ermöglicht durch selbstbeschreibende Nachrichten die Aushandlung eines geeigneten Formats zur Laufzeit. Dadurch können Client-Entwickler selbst entscheiden, was ihren Anforderungen am besten gerecht wird. Dies bedeutet ebenfalls, dass es keine Unterscheidung zwischen menschlichen und maschinellen Dienstkonsumenten gibt.

Zum anderen ermöglicht Hypermedia bereits im Dienst, Kontrollflussinformationen zu kodieren.

1.3 Ziel der Arbeit

Im Rahmen dieser Arbeit sollen die Auswirkungen des Architekturstils REST auf die Anwendungsentwicklung betrachtet werden. Dabei stehen solche Anwendungen im Fokus, die innerhalb einer dienstorientierten Architektur eines Unternehmens verwendet werden. Mittels eines Fallbeispiels soll aufgezeigt werden, wie sich die Anwendungsentwicklung je nach gewähltem Architekturstil unterscheidet und wie die von REST neu aufgeworfenen Problematiken insbesondere bei der Komposition von Diensten gehandhabt werden können. Hierzu werden mehrere aktuelle Arbeiten aus dem Zeitraum von 2008 bis 2010 diskutiert.

Aus einer pragmatischen Perspektive auf dieses Thema sollen auch die zum Zeitpunkt der Entstehung dieser Arbeit verfügbaren REST-Frameworks diskutiert und eine Empfehlung hinsichtlich ihrer Verwendung gegeben werden.

1.4 Aufbau der Arbeit

Kapitel 1 stellt die Einleitung in das Thema dieser Arbeit dar. Dabei sind mehrere Probleme aufgezeigt, die aus der Verwendung von REST zur Anwendungsentwicklung resultieren.

Im Anschluss werden in Kapitel 2 Grundlagen des Themenbereichs erläutert. Dazu gehören Begriffsabgrenzungen, Beschreibungen von verschiedenen Modellierungssprachen und eine Zusammenfassung des Architekturstils REST.

Zur Vertiefung des Verständnisses dieser Hintergrundinformationen werden diese in Kapitel 3 auf ein Fallbeispiel angewandt.

Die dazu vorgenomme Implementierung wird im folgenden Kapitel 4 dazu verwendet, die Problematik der Umsetzung des Hypermedia-Constraints zu diskutieren. Dazu werden mehrere Frameworks sowie die JAX-RS (Java API for RESTful Web Services) Spezifikation als auch das HTTP (Hypertext Transfer Protocol) betrachtet.

Kapitel 5 geht auf die Komposition von RESTful Services ein. Hierbei werden mehrere Ansätze von Alarcon et. al. sowie Pautasso vorgestellt und ihr Umgang mit den in der Einleitung beschriebenen Problemen herausgearbeitet. Abschließend wird eine Empfehlung für die Verwendung der Ansätze abgegeben.

An diese Thematik angeschlossen, wird in Kapitel 6 auf die Problematik der Service Discovery eingegangen und gezeigt, dass bei der Implementierung einer SOA die semantische Lücke sowohl bei REST als auch jedem anderen Architekturstil nicht automatisiert

überwunden werden kann.

Die vorliegende Arbeit wird in Kapitel 7 abgeschlossen, indem die Erkenntnisse der Auseinandersetzung mit den untersuchten Technologien, Frameworks, Kompositionsansätzen den zu Beginn vorgestellten Problemen zugeordnet werden. Im Ausblick werden mögliche weitere Entwicklungen aufgezeigt sowie eine Einschätzung der weiteren Verbreitung von REST in der Praxis abgegeben.

2 Hintergrundinformationen

2.1 Anforderungen an Anwendungen

2.1.1 Lose Kopplung

Der Begriff der losen Kopplung wird in der Softwareentwicklung sehr häufig verwendet und gilt bei der Entwicklung einer Anwendung als erstrebenswert. Hock-koon et al. verstehen unter loser Kopplung die Reduktion von Abhängigkeiten, um eine größere Flexibilität zu erhalten (siehe [19]). Diese sehr generische Betrachtung wird durch Hock-koon et. al. in Bezug auf dienstorientierte Architekturen (siehe Abs. 2.2.1) in drei Arten von Kopplung aufgetrennt: semantische, syntaktische und physikalische Kopplung.

Die *semantische Kopplung* resultiert aus dem Wissen der Fachseite über die Anwendungsdomäne. Dadurch, dass in abstrakten Diensten dieses Wissen explizit gemacht wird, werden die Dienste hinsichtlich ihrer Semantik aneinander gekoppelt. Hock-koon et al. differenzieren drei Grade der semantischen Kopplung: starke, lose und unwirksame („non predominant“) Kopplung.

Die *syntaktische Kopplung* fokussiert auf die Abhängigkeiten zwischen abstrakten und konkreten Diensten. Eine starke syntaktische Kopplung liegt dann vor, wenn ein abstrakter Dienst durch lediglich einen einzigen konkreten Dienst implementiert wird. Die Funktionsweise des abstrakten Dienstes ist damit vom Vorhandensein des konkreten Dienstes abhängig. Durch die Einführung weiterer konkreter Dienste, die gegeneinander ausgetauscht werden können, wird die syntaktische Kopplung reduziert.

Die *physikalische Kopplung* zeichnet sich zwischen konkreten Diensten ab. Hier kommt es laut Hock-koon et al. zu einem starken Aufkommen von horizontaler Kommunikation. Bei dieser handelt es sich um einen Nachrichtenaustausch zwischen den konkreten Diensten ohne die Verwendung eines Mediators. Eine physikalische Entkopplung wird dadurch erreicht, dass auf vertikale Kommunikation geachtet wird, indem die konkreten Dienste in einer abstrahierten Kompositionsschicht durch eine zusätzliche Instanz kontrolliert werden. In einer Orchestrierung würde diese Rolle dem Dirigenten entsprechen.

2.1.2 Interoperabilität

Xu et al. unterstreichen in ihrer Arbeit [47] die starke Bedeutung von Interoperabilität für die Domäne des Geschäftsprozessmanagements. Standardisierung ist eine Möglichkeit, um Interoperabilität zu erreichen. Ein Beispiel für Standardisierungen für die Modellierung von Geschäftsprozessen (siehe Abs. 2.4.1) stellt die BPEL (Business Process Execution Language; siehe Abs. 2.4.2) dar, wobei hier die Prozesskoordination im Vordergrund steht. Interoperabilität geht jedoch über die reine Koordination hinaus. So fällt die Frage, ob Dienste überhaupt miteinander koordiniert werden können, ebenfalls unter den Begriff der Interoperabilität. Xu et al. differenzieren drei Ebenen, auf denen sich Interoperabilität betrachten lässt.

Interoperabilitätsebene	**WS***	**REST**
syntaktisch	WSDL	uniformes Interface
	XMLSchema	typisierte Datenformate
protokollbasiert	manuell im Prozessmodell	durch Hypermedia
semantisch	informelle Absprache	
	OWL-S	RESTfulGrounding

Tabelle 1: Unterstützung der Interoperabilität abhängig vom Architekturstil

- Auf *Signaturebene* zeichnet sich Interoperabilität dadurch aus, dass die Signaturen von verwendeten Operationen syntaktisch zueinander kompatibel sind. Die WSDL (siehe Abs. 2.3.1) wird dazu verwendet, um diese syntaktische Kompatibilität herzustellen. Bei REST wird die signaturbezogene Interoperabilität durch das uniforme Interface unterstützt. Die Verwendung von Metadaten innerhalb ausgetauschter Repräsentationen können dazu beitragen, dies weiter zu fördern. Durch die aushandelbaren Datenformate wird eine Typisierung der in den Nachrichten enthaltenen Repräsentationen möglich. Dabei sollte angemerkt werden, dass Eingabeparameter wie die *Messages* der WSDL bei REST-Architekturen in der Regel in den ausgetauschten Ressourcenrepräsentationen kodiert sind.

- Die *Protokollebene* der Interoperabilität legt fest, in welcher Abfolge bestimmte Interaktionen mit einem Dienst möglich sind. REST setzt diese Interoperabilität mittels des Hypermedia-Constraints um. Die WSDL oder SOAP stellen hierfür keine Lösungen bereit. Die protokollbasierte Interoperabilität muss hier im Prozessmodell definiert und durch den Client wie z.B. eine Prozessengine überwacht werden.

- *Semantische* Interoperabilität zeichnet sich dadurch aus, dass die Anbieter und Konsumenten von Diensten ein gemeinsames Verständnis über die Bedeutung der Operationen und Daten bzw. Ressourcen und Verknüpfungen besitzen. Die Semantik der Methoden des uniformen Interfaces muss bei REST zwar auch von beiden beteiligten Seiten geteilt werden, jedoch wird diese Semantik im verwendeten Protokoll (z.B. HTTP) definiert und muss nicht individuell für jede Anwendung neu ausgehandelt werden. OWL-S stellt eine Ontologie zur Identifikation, Komposition und Ausführung von durch WSDL beschriebenen Webservices dar (siehe [24]), wobei die semantische Beschreibung nicht für zwischen den Diensten ausgetauschten Informationen ausgelegt ist. Daher muss für domänenspezifische semantische Interoperabilität z.B. auf informelle Absprachen zurückgegriffen werden. Solche Absprachen sind auch bei REST-Architekturen möglich, wobei durch „RESTfulGrounding“ ein Analogon zu OWL-S für WADL mit denselben Einschränkungen existiert (siehe [15]).

2.2 Dienstorientierung

2.2.1 Dienstorientierte Architektur

Der Begriff der dienstorientierten Architektur (Service Oriented Architecture, SOA) wird in der Literatur unterschiedlich verwendet. Dies liegt an den unterschiedlichen gebräuchlichen Definitionen.

Mittal versteht unter dem Begriff der dienstorientierten Architektur hingegen *„eine Menge von Werkzeugen, Technologien, Rahmenwerken und Erfahrungswerten, die es ermöglichen, schnell und einfach Dienste zu implementieren“* (siehe [23]).

Hock-koon und Oussalah definieren SOA als ein Muster der Softwareentwicklung, welches Mechanismen für eine homogene Exposition und Nutzung von heterogenen Ressourcen bereitstellt (siehe [19]). Sie sehen in der Dienstkomposition einen solchen Mechanismus, da er vorhandene Ressourcen als neue abstraktere Dienste bereitstellt. Außerdem betonen Hock-koon et al., dass die lose Kopplung von Ressourcen bei der Implementierung einer SOA zentral ist.

Tilkov bezeichnet SOA „als technologieunabhängiges Ziel der Unternehmens-IT“ (siehe [45]). Dabei steht die Entwicklung von unabhängigen entkoppelten Diensten im Vordergrund, die nach den Bedürfnissen des Unternehmens ausgerichtet sind. Durch die Interaktion der einzelnen Dienste miteinander sollen die Geschäftsprozesse des Unternehmens unterstützt werden.

Unabhängig von den unterschiedlichen SOA-Begriffsdefinitionen ist festzuhalten, dass es sich bei SOA nicht um einen Standard oder um ein konkretes Software-Produkt handelt. Darüber hinaus stellt die Interoperabilität von Diensten ein zentrales Kriterium für die erfolgreiche Implementierung einer SOA dar.

Im weiteren Verlauf dieser Arbeit wird der SOA-Begriff gemäß der Definition Tikov verwendet.

Im Zusammenhang mit diesen Definitionen ist es wichtig, die Bedeutung des „Dienst“-Begriffs zu klären. Unter einem Dienst wird laut Hagen Overdick ein *„Mechanismus verstanden, der Zugang zu einer oder mehr Funktionalitäten bietet, wobei die möglichen Konsumenten dem Dienst nicht bekannt sein müssen.“* (siehe [31]).

Pascalau et al. definieren den Begriff des Diensts wie folgt:

> *„Ein Dienst ist eine Handlung oder Leistung, die von einem Beteiligten an einen anderen gerichtet ist. Obwohl der Prozess an ein physikalisches Produkt gebunden sein kann, ist die Leistung immateriell und führt normalerweise nicht zum Besitz von einzelnen Produktionsfaktoren.“* (übersetzt aus [32])

Da diese beiden Definitionen sich gegenseitig erweitern, werden in dieser Arbeit beide Erläuterungen verwendet.

2.2.2 Dienstkomposition

Unter Dienstkomposition wird die Kombination von Diensten mit dem Ziel, höherwertige abstraktere Dienste zu erstellen, verstanden. Laut Pautasso unterliegen diese zusammengesetzten (komponierten) Dienste selbst einer rekursiven Natur, die ihre Rekomposition ermöglicht (siehe [33]). D.h. Dienste, die das Ergebnis einer Komposition sind, können wiederum selbst Teil einer weiteren Komposition sein. Es gibt mehrere Arten von Dienstkompositionen, wobei Peltz (siehe [36]) hierbei zwei Aspekte unterscheidet: Orchestrierung und Choreografie.

Unter einer *Orchestrierung* wird *„ein ausführbarer Geschäftsprozess verstanden, der sowohl mit internen als auch externen Diensten interagieren kann“* (übersetzt nach [36]). Innerhalb der Orchestrierung sind Informationen über die Ablaufreihenfolge von Aktivitäten und implizit Informationen über die Logik der Anwendungsdomäne enthalten. Zentrales Unterscheidungsmerkmal in Bezug auf die Choreografie ist, dass bei der Orchestrierung eine zentrale Steuerungseinheit vorhanden ist - ähnlich zu einem Dirigenten bei einem Orchester. Tilkov vergleicht diese Steuerungseinheit (Orchestration Engine) mit einer Laufzeitumgebung ähnlich zu einem Interpreter für Skriptsprachen (siehe [45]).

Bei einer Orchestrierung wird die horizontale Kommunikation zwischen Diensten reduziert und in einer höhere Abstraktionsebene verlagert. Dadurch entsteht eine vertikale Kommunikation zwischen den in der Komposition verwendeten Diensten und dem Mediator (vgl. Abs. 2.1.1) - in diesem Fall der *Orchestration Engine*.

Eine *Choreografie* verzichtet auf das zentrale Steuerungselement und lässt die Dienste selbst miteinander interagieren. Es besteht somit die Gefahr, dass es zu einer physikalischen Kopplung (vgl. Abschnitt 2.1.1) von Diensten kommt, da horizontale Kommunikation im Vordergrund steht. Pascalau et al. stellen heraus, dass sog. *Mashups* ein Beispiel für browser-basierte Dienstchoreografie sind (siehe [32]). Pautassos Betrachtung von Mash-ups als *„neue Art von Webanwendungen, welche Datenquellen und Webservices anderer Anbieter miteinander kombiniert“* (übersetzt aus [34]), untermauert diese Aussage.

2.2.3 Dienstsemantik

Unter dem Begriff der Dienstsemantik wird im weiteren Verlauf der Arbeit die Bedeutung dessen verstanden, was ein Dienst auf einer domänenspezifischen Konzeptebene leistet. Bei der Dienstsemantik geht es daher nicht um eine syntaktische Betrachtung von Diensten.

Andreas Schmidt schlüsselt Dienstsemantik in mehrere Bereiche auf (siehe [41]). Die *Parameter* eines Dienstes besitzen eigene Bedeutungen, die zur Erreichung von Interoperabilität verstanden werden müssen. Die *Funktionalitäten* des Dienstes besitzen eine eigene Semantik, die zur Entdeckung von Diensten erkennbar sein muss. Darüber hinaus weist Schmidt auf die Bedeutung der Kriterien für Dienstgüte hin.

2.3 Dienstbeschreibung

2.3.1 Web Services Description Language

Die Web Services Description Language (WSDL) ist eine auf XML aufbauende Sprache zur Beschreibung von Webserviceschnittstellen (siehe [18]). In Version 1.1 wird SOAP als einziges Binding[1] in Kombination mit der GET- sowie POST-Methode des HTTP aufgeführt (siehe [7]). Aktuell liegt die WSDL in Version 2.0 vor. Sie führt Erweiterungen ein, die sich zur Beschreibung von RESTful HTTP Services unter Einschränkungen verwenden lassen (siehe [45] S. 144). So ist es durch die WSDL nicht möglich, verfügbare Formate einer Ressourcenrepräsentation frei zu bestimmen oder die Beziehungen zwischen Ressourcen zu beschreiben.

Betrachtet man die WSDL im Zusammenhang mit BPEL, so ist hervorzuheben, dass die Erweiterungen von WSDL 2.0 nicht verwendet werden können. Eine Orchestrierung von RESTful HTTP Services mittels BPEL ist nicht möglich, da BPEL explizit auf die WSDL in Version 1.1 angewiesen ist [30].

Ein WSDL-Dokument definiert eine Menge von Datentypen, Nachrichten, PortTypes, Bindings, Ports und Services. Zusammengenommen ergeben diese Elemente eine Dienstbeschreibung, die es entweder ermöglicht einen Client mit dem entsprechendend beschriebenen Dienst kommunizieren zu lassen oder die Kommunikationsschnittstelle eines Diensts aus der Beschreibung generieren zu lassen. Das WSDL-Dokument dient somit als Vertrag über die verwendete Kommunikationssyntax.

2.3.2 Web Application Description Language

Die Web Application Description Language (WADL) stellt eine Beschreibungssprache dar, die für Dienste auf der Basis von RESTful HTTP ausgelegt ist. Mir ihr lässt sich ein statischer Vertrag zwischen Diensten und den Konsumenten formulieren. Wie bei der WSDL werden auch WADL-Dokumente in XML-Syntax geschrieben (siehe [45]). Tilkov stellt heraus, dass sich analog zu WSDL und SOAP basierten Webservices auch eine WADL-Beschreibung eines Dienstes aus dessen Implementierung generieren lässt. Hierzu werden z.B. im Falle einer Java-Implementierung die JAX-RS (siehe Abs. 4.2) Annotationen aus dem Quellcode ausgelesen, um HTTP-Methoden, URI-Templates und Datenformate festzulegen (siehe [45]).

Die WADL besitzt keine Sprachelemente, die sich zur Beschreibung der von Hypermediainformationen innerhalb oder außerhalb von Ressourcenrepräsentationen einsetzen lassen. Stattdessen wirkt WADL mit der expliziten Benennung von URIs oder URI-Templates dem HATEOAS-Prinzip (siehe Abs. 2.5) von REST entgegen, da URIs abgesehen von einem Einstiegspunkt in den Dienst erst zur Laufzeit entdeckt werden sollen.

[1] Ein Binding legt das in der Interaktion mit einem Dienst zu verwendende Übertragungsprotokoll sowie den zu verwendenden Datentyp fest.

Der enge Zusammenhang zwischen der WADL und der JAX-RS-Spezifikation lässt sich dadurch erklären, dass beide durch Marc Hadley mitentwickelt wurden.

2.4 Geschäftsprozessmodellierung

2.4.1 Geschäftsprozess

In der Literatur existieren viele Definitionen für den Begriff des Geschäftsprozesses, so dass bereits Sammelwerke entstanden sind. Lindsay et al. [21] stellen dabei heraus, dass bei der Modellierung von Geschäftsprozessen die *„internen Elemente“* der Prozesse repräsentiert werden sollen. Derartige Elemente sind u.a. Aktivitäten, Aktivitätsabhängigkeiten, Rollen, Akteure und Ziele.

Unter einem Geschäftsprozess wird nach Masak ein *„betrieblicher Prozess, der zur Erstellung der Organisationsleistung beiträgt“* [22], verstanden. Ein Prozess setzt sich dabei aus mehreren Aktivitäten zusammen. Unter einer Aktivität wird ein diskreter Schritt innerhalb des Prozesses verstanden, wobei dieser von Menschen oder Diensten durchgeführt werden kann. Wird ein Geschäftsprozess ausgeführt, so ändert sich dadurch immer der Zustand der Organisation.

Masak stellt ebenfalls heraus, dass ein Geschäftsprozess formal beschrieben werden kann. Hierzu kann z.B. auf die BPMN (Business Process Modeling) Notation verwiesen werden. Weitere Charakteristika für Geschäftsprozesse sind eindeutige Ein- und Ausgaben sowie ein definierter Kontext.

Eine weitere Eigenschaft von Geschäftsprozessen ist, dass diese von der Umwelt ausgelöst werden und auch in der Umwelt enden. Das kann bedeuten, dass ein Geschäftsprozess z.B. von einem Kunden gestartet wird und auch wieder beim Kunden endet.

Laut Mongiello und Castelluccia ist ein Geschäftsprozess *„ein komplexer Webservice, der aus der Komposition von bereits verfügbaren und dynamisch integrierten Webservices entspringt.“* [24]. Sie reduzieren Geschäftsprozesse damit auf eine technische Ebene, die keine von Menschen durchgeführten Aktivitäten vorsieht. Ihr Fokus liegt damit bei automatisch ausführbaren Prozessen. Damit reduzieren sie den Begriff des Geschäftsprozesses mit dem Workflow-Begriff gleich.

Nach Cichocki et al. [8] unterscheiden sich die Begriffe des Prozesses und des Workflows jedoch wie folgt: Ein Prozess ist eine Beschreibung der Arbeitsaktivitäten und ihrer Ausführungsabfolge. Er definiert strukturelle Aspekte und organisatorische Funktionen. Prozesse lassen sich in Workflows abbilden, wobei ein Workflow eine Prozessbeschreibung ist, die automatisiert ausgeführt werden kann. Dabei wird durch die Workflows spezifiziert, *wie* die einzelnen Aktivitäten ausgeführt werden. Ob die Aktivitäten automatisiert durch technische Systeme oder durch manuelle Tätigkeiten ausgeführt werden, ist nicht festgelegt. In der Regel kommen jedoch technische Systeme zum Einsatz.

Im weiteren Verlauf dieser Arbeit wird der Geschäftsprozessbegriff nicht synonym zu dem eines Workflows, sondern gemäß der Beschreibung von Cichocki et al. verwendet.

2.4.2 Business Process Execution Language

Die Business Process Execution Language (BPEL bzw. WS-BPEL) ist eine auf XML aufgebaute Sprache zur Steuerung von ausführbaren Geschäftsprozessen (Workflows) (siehe [5]). Dabei liegt der Fokus auf Webservices, die mittels eines WSDL Dokuments spezifiziert sind (siehe [33]). Mittels BPEL lassen sich Webservices durch eine Prozessausführungsumgebung[2] gesteuert orchestrieren. Dabei werden die Operationsaufrufe an den verwendeten Diensten von der Ausführungsumgebung kontrolliert. Ziel ist es, durch die Orchestrierung eine höherwertige, bzw. abstraktere Funktionalität über eine eigene Serviceschnittstelle anzubieten (vgl. Dienstkomposition in Abs. 2.2.2).

Tilkov stellt in Bezug auf die Prozessausführungsumgebung eine Analogie zu einem Interpreter auf, da die durch BPEL beschriebenen Prozesse eine Abfolge von Operationsaufrufen analog zu einer Skriptsprache darstellen. Im Unterschied zu einem Interpreter stellt er jedoch die Möglichkeit der asynchronen Verarbeitung durch die Prozessausführungsumgebung heraus (siehe [45]).

2.4.3 Business Process Model and Notation

Die Business Process Model and Notation[3] (BPMN) stellt einen Standard zur Modellierung von Geschäftsprozessen dar. Sie kann als Sprache zur Spezifikation und Dokumentation von Geschäftsprozessen verstanden werden. Im Gegensatz zu BPEL besitzt die BPMN eine grafische Repräsentationsform. Darüber hinaus legt der BPMN 2.0 Standard auch ein Datenformat für den Austausch von Modellen fest (siehe [5]). Darüber hinaus ist die BPMN jedoch implementierungsunabhängig und erlaubt die Beschreibung von Choreografien durch den Nachrichtenaustausch zwischen einzelnen Systemen. Gleichzeitig ist es jedoch auch möglich, Orchestrierungen in der BPMN zu visualisieren (siehe [37]).

Neben den reinen Konstrukten zum Aufbau der Modelle steht bei der BPMN außerdem noch die Ausführbarkeit der Modelle im Vordergrund. Der Standard definiert eine Semantik, wie BPMN-Modelle in einer Ausführungsumgebung abzuarbeiten sind. Die BPMN-Spezifikation hält jedoch trotz ihres Anspruchs auf Ausführbarkeit der Modelle zusätzlich auch Regeln zur Umwandlung der Modelle in das BPEL-Format bereit (siehe [5]).

Aktuell wird die BPMN vorwiegend dazu verwendet, um Geschäftsprozesse zu dokumentieren. Dies Ausführbarkeit und Vollständigkeit der Modelle steht weniger im Vordergrund. Dies geht aus den Arbeiten von Allweyer [5] und Ouyang et al. [25] hervor. Hinsichtlich der Motivation zur Entstehung sind sich Allweyer [5] und Ouyang et. al. [25] nicht einig: Allweyer sieht in der BPMN eine Notation, die auch die Ausführbarkeit der Modelle ermöglicht, während Ouyang et al. sagen, dass die BPMN eher für die Konzeption von Geschäftsprozessen gedacht ist - d.h. nicht für die konkrete Ausführung. Die ausführbaren Modelle werden nach Ouyang et al. nach wie vor in BPEL beschrieben. Dies spiegelt sich in der Implementierungsunabhängigkeit der BPMN wider. Der Ablauf eines mittels BPMN

[2] wie z.B. Apache ODE oder JBoss RiftSaw

[3] Früher auch „Business Process Modeling Notation“ genannt [10]

modellierten Prozesses ist zwar durch die Ausführungssemantik festgelegt, jedoch fehlen Verbindungen zu einer Implementierung der ausführenden Instanz, die etwa Funktionalitäten eines Webservice aufruft.

2.5 Representational State Transfer

Representational State Transfer (REST) ist ein Architekturstil für verteilte Hypermedia-Systeme, der 2000 von Roy Thomas Fielding beschrieben wurde.

Für das Verständnis von REST ist der Begriff der *Ressource* zentral: Unter einer Ressource wird eine Abstraktion für Informationen verstanden (siehe[14]). Informationen müssen sich in diesem Zusammenhang nicht auf Dateien oder Datenbankeinträge beschränken. Stattdessen kann eine Ressource auch Diensten (bzw. Dienstleistungen), realen Objekten oder Prozessen (siehe [45]) entsprechen. Die Zustände von Ressourcen können sich über die Zeit verändern und sind unabhängig von einer Repräsentation.

REST sind fünf Prinzipien zugrunde gelegt, die von Anwendungen berücksichtigt werden müssen, um als „RESTful“ bezeichnet werden zu können. Das erste Prinzip stellt die Referenzierung von Ressourcen durch eine oder mehrere *eindeutige URIs* (Unified Resource Identifier) dar. Dies bedeutet jedoch nicht, dass im Umkehrschluss jeder URI eine bereits existierende Ressource zugeordnet sein muss. Durch eine URI können auch Ressourcen adressiert werden, die noch nicht oder nicht mehr existieren.

Ein weiterer zentraler Begriff ist *Hypermedia*. Fielding definiert ihn in seiner Dissertation [14] als *„das Vorhandensein von Anwendungskontrollinformationen innerhalb oder als Abstraktionsschicht über der Repräsentation von Informationen“* (sinngemäß übersetzt). Dies bedeutet, dass der Nutzungsablauf einer nach REST entwickelten Anwendung durch Hypermedia gesteuert wird. Dadurch wird die Menge aller möglichen Nutzungsabläufe durch die Verknüpfung von Ressourcen implizit in der Anwendung abgelegt.[4]. Fielding bezeichnet Hypermedia daher auch als *„engine of application state“* (siehe S. 82 in [14]). Dieses Prinzip wird in der Literatur wie z.B. durch Tilkov oft mit *HATEOAS* abgekürzt (siehe S.69 in [45]).

Ein weiteres Prinzip von REST stellt die Verwendung eines *generischen einheitlichen Interfaces* dar. Hierdurch soll die Architektur eines Systems vereinfacht und die Sichtbarkeit von Interaktionen zwischen Systemkomponenten erhöht werden (siehe [14]). REST legt jedoch keine konkrete Ausprägung des uniformen Interfaces fest. D.h. die zur Verfügung stehenden Methoden werden erst durch die Implementierung wie z.B. das HTTP festgelegt. Fielding argumentiert, dass dieses Schnittstellendesign zur Effizienzsteigerung in großen Hypermedia-Anwendungen konzipiert ist.

Wie bereits erwähnt, werden innerhalb des REST-Architekturstils nicht Ressourcen, sondern deren Ressourcenrepräsentationen in *selbstbeschreibende Nachrichten* ausgetauscht.

[4] Vergleicht man dies nun mit Diensten, die operationsorientiert sind wie etwa auf der Implementierungsbasis von SOAP, so wird deutlich, dass dort Hypermedia innerhalb des implementierenden Dienstes fehlt. Stattdessen ist eine zusätzliche Abstraktionsschicht in Form von Orchestierung möglich, die den Anwendungsfluss kontrolliert.

Durch dieses Prinzip soll laut Fielding eine Zwischenverarbeitung von Interaktionen der Systemkomponenten ermöglicht werden. Die Selbstbeschreibung erfolgt durch Metadaten (siehe S. 90 [14]).

Abschließend sei das Prinzip der *statuslosen Kommunikation* in einer REST-Architektur aufgeführt. D.h. die Anwendungen innerhalb einer solchen Architektur müssen alle zur Verarbeitung einer Nachricht notwendigen Informationen innerhalb der Nachrichten austauschen. Ein Dienst darf daher keine Annahmen darüber treffen, in welchem Zustand sich ein Konsument befindet, um dessen Anfrage zu bearbeiten. Das Prinzip verbietet jedoch nicht, dass ein Dienst keinen Status speichern dürfte. Dinge, die einen Status besitzen, werden zu Ressourcen und können mittels einer Repräsentation statuslos mit der Anwendung ausgetauscht werden (siehe S. 15 in [45]).

An verschiedenen Stellen der Literatur findet sich die Behauptung, dass REST direkt durch HTTP 1.1 widergespiegelt wird (siehe [48]). Diese Aussage sollte jedoch kritisch betrachtet werden, da das HTTP - je nach Version - unterschiedlich mit Hypermedia umgeht. Dies wird in Abschnitt 4.1 näher disktutiert.

3 Fallbeispiel

Dieses Kapitel soll die Prinzipien von REST an Hand eines Fallbeispiels verdeutlichen, indem diese auf eine Implementierung angewandt werden. Um Unterschiede zu den bisher in SOAs verbreiteten SOAP und WSDL basierten Diensten aufzeigen zu können, wird zusätzlich das Vorgehen bei einer Implementierung mit den WS*-Technologien betrachtet. Außerdem werden die Problematiken aus Kapitel 1.2 an Hand des Fallbeispiels aus einer praxisorientierten Perspektive aufgezeigt.

3.1 Beschreibung des Szenarios

Um ein möglichst realistisches Fallbeispiel zu besitzen wird in dieser Arbeit die Domäne eines Telekommunikationsunternehmens verwendet. Die folgenden Beschreibungen sind frei erfunden und spiegeln kein konkretes real existierendes Unternehmen wider. Das Fallbeispiel orientiert sich an einem Geschäftsprozess, der innerhalb eines Unternehmens dieser Domäne sehr zentral ist: Die Beratung eines Kunden und der anschließende Abschluss eines privaten Neukunden-Mobilfunkvertrags.

3.1.1 Prosa

Das auslösende Ereignis für diesen Prozess ist das Betreten einer Geschäftsfiliale durch den Kunden mit dem Ziel, einen neuen Mobilfunkvertrag abzuschließen. Er spricht einen Mitarbeiter an und teilt ihm seinen Wunsch mit. Der Mitarbeiter nennt dem Kunden die verfügbaren Tarife und lässt den Kunden gemäß den persönlichen Bedürfnissen einen Tarif auswählen. Nachdem sich der Kunde für einen Tarif entschieden hat, ruft der Mitarbeiter ein leeres Vertragsformular auf. Hierbei wird im Vertriebsportal ein leerer Vertrag angelegt. Anschließend befragt der Mitarbeiter den Kunden nach seinen persönlichen Daten (Name, Alter, Geburtsort, Adresse, Telefonnummer, Emailadresse etc.) und gibt diese zusammen mit dem vom Kunden gewünschten Tarif in das Formular des Privatkundenvertriebsportals ein.

Daraufhin stellt der Mitarbeiter die Frage, ob der Kunde eine alte Rufnummer von einem vorherigen Mobilfunkvertrag in den neuen Vertrag mitnehmen möchte. Für den Fall, dass der Kunde dies wünscht, muss der Mitarbeiter einen Nummerntransfer beim alten Mobilfunkbetreiber des Kunden beantragen. Hierbei handelt es sich jedoch um einen komplexeren Sub-Prozess, der in diesem Fallbeispiel nicht weiter betrachtet wird. Wird kein Nummerntransfer gewünscht, so schickt der Mitarbeiter das Formular mit den persönlichen Daten und den Tarifinformationen direkt ab. Dies hat zur Folge, dass die Vertragsdaten im Vertriebsportal innerhalb des bereits angelegten Vertrags ergänzt werden. Der Vertrag wechselt seinen Zustand nicht und wird weiterhin als neu betrachtet. Sobald die Daten gespeichert sind, kann der Mitarbeiter einen Mustervertrag drucken und an den Kunden aushändigen, damit dieser die Daten kontrolliert. Für den Fall, dass die Daten nicht korrekt sind, korrigiert der Kunde diese Fehler und der Mitarbeiter muss die korrigierten Daten erneut in das

Formular eingeben und die Vertragsdaten im Vertriebsportal abspeichern. Dieser Vorgang kann mehrfach wiederholt werden, bis die Daten korrekt sind. Sobald dies der Fall ist, bestätigt der Kunde die Korrektheit der Daten. Mit diesen Informationen ruft der Mitarbeiter nun die Bonitätsinformationen mit Hilfe des Vertriebsportals ab. Das Vertriebsportal selbst greift dafür auf einen externen Rating-Dienst zurück. Sobald der Dienst mit den Bonitätsinformationen geantwortet hat, werden diese vom Vertriebsportal dem Vertrag zugeordnet und der Mitarbeiter benachrichtigt.

Ist die Bonitätsbewertung des Kunden durch den Rating-Dienst schlecht ausgefallen, so wird der Kunde abgewiesen und der Prozess ist beendet. Eine wiederholte Prüfung der Bontität ist möglich, so dass der Kunde ggf. zu einem späteren Zeitpunkt doch als zahlungsfähig eingestuft wird. Wird der Kunde direkt als zahlungsfähig eingestuft, so erhält er ein Formular, in dem er seine Kontodaten bekannt gibt und eine Einzugsermächtigung für das Telekommunikationsunternehmen erteilt. Dieses Formular gibt der Kunde ausgefüllt an den Mitarbeiter zurück, welcher die Bankinformationen dem Vertrag zuordnet. Hierzu werden die entsprechenden Daten an das Vertriebsportal übergeben. Zeitgleich wird der Vertrag vom Kunden unterschrieben und an den Mitarbeiter weitergegeben. Außerdem erhält der Kunde ein eigenes Exemplar des Vertrags. Sobald der Mitarbeiter den Vertrag und die Bankdaten erhalten hat, bestätigt er diesen Empfang, indem er den Vertrag im Vertriebsportal aktiviert. Dies hat zur Folge, dass das Vertriebsportal die Abrechnungsstelle über den neuen Vertrag benachrichtigt. Im nächsten Schritt, weist der Mitarbeiter dem Vertrag eine neue SIM-Karte (bzw. IMEI, ICCID) zu. Diese Zuweisung teilt er ebenfalls dem Vertriebsportal mit, damit dieses die SIM-Karte freischalten lässt. Abschließend händigt der Mitarbeiter die SIM-Karte an den Kunden aus, der nun die Filiale verlässt.

3.1.2 BPMN-Modell

Der im Text beschriebene Prozess lässt sich mittels eines formalen Modells in der BPMN ausdrücken. Da das Modell zu breit für die Darstellung auf einer einzelnen Seite ist, ist dieses Modell im Anhang A.3 zu finden. Ziel ist es dabei, in diesem Fall den Prozess durch eine visuelle Präsentation leichter verständlich zu machen. Die Erstellung eines ausführbaren Prozessmodells ist nicht beabsichtigt. Dies wird in Kapitel 5 näher diskutiert.

Innerhalb des Modells werden vier verschiedene Pools verwendet: Kunde, Telekommunikationsunternehmen, Vertriebsportal und Schufa. Die einzelnen Bahnen eines Pools repräsentieren die am Prozess beteiligten Organisationseinheiten und Rollen (siehe [5]). Innerhalb der Kunden-Bahn befinden sich alle Aktivitäten und Ereignisse, die den Kunden betreffen. Die Bahn des Telekommunikationsunternehmens beinhaltet alle Aktivitäten und Ereignisse, die dem Mitarbeiter in der Filiale des Unternehmens zugeordnet sind. Bei den beiden verbleibenden Bahnen des Vertriebsportals und des Rating-Dienstes handelt es sich um am Geschäftsprozess beteiligte IT-Systeme. Um die Übersicht der Darstellung zu wahren werden Backend-Systeme wie z.B. zur Verwaltung von Netzzugängen der Mobilfunkkunden nicht betrachtet, die an der tatsächlichen Freischaltung eines Mobilfunkzugangs

beteiligt sind.

Der modellierte Prozess besitzt zwei mögliche Endzustände: Entweder erhält der Kunde seinen Vertrag und die zugehörige SIM-Karte oder er wird als Kunde vom Telekommunikationsunternehmen abgelehnt und der Vertrag bleibt in einem nicht-aktiven Zustand.

3.1.3 Zustandsmodell

Im vorgestellten Beispielprozess ist der „Vertrag“ das primäre Geschäftsobjekt. Ein Vertragsobjekt kann während seiner Existenz verschiedene Zustände durchlaufen. Diese sind in Abbildung 1 dargestellt. Wird ein Vertrag angelegt, so befindet er sich im Zustand „new“. In der Domäne des Fallbeispiels bedeutet dies, dass es sich hierbei um einen Vertrag handelt, der gerade erstellt wird. Sobald das Ergebnis der Bonitätsprüfung des Kunden vorliegt, führt dieses zu einem Zustandswechsel des Vertrags: Im Falle ausreichender Bonität wechselt der Zustand zu „creditworthy“ bzw. im Falle mangelhafter Bonität zu „not creditworthy“. Die Prüfung kann mehrmals wieder holt werden, so dass ein Wechsel zwischen den beiden zuletzt genannten Zuständen möglich ist. Zu einem Vertragsabschluss kann es nur kommen, wenn die Kreditwürdigkeit des Kunden positiv beurteilt ist. Wird der Vertrag aktiviert, so wechselt der Zustand des Vertrags zu „alive“. Ein Rückschritt zu einem der vorherigen Zustände ist nicht möglich, da dies in der Anwendungsdomäne einen Vertragsbruch darstellen würde. Die Kündigung oder das Auslaufen eines Vertrags ist bewusst nicht modelliert, um das Modell kompakt und übersichtlich zu halten.

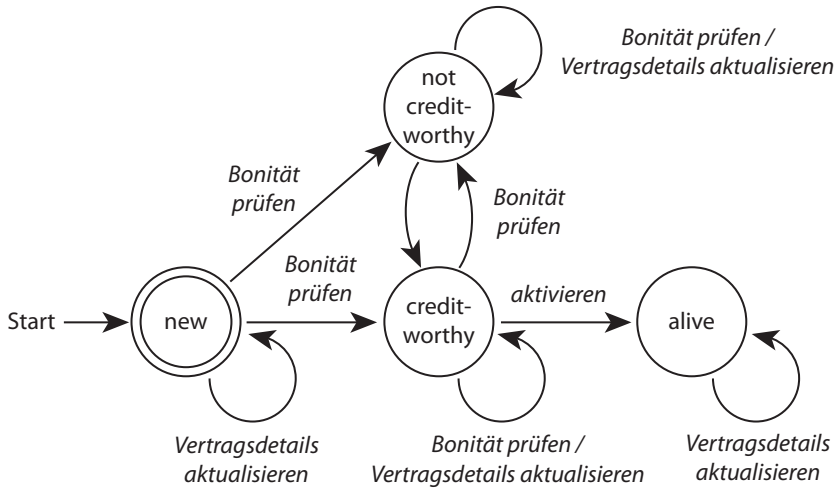

Abbildung 1: Zustandsmodell für das Geschäftsobjekt „Vertrag“

In Bezug auf REST stellen Verträge eine Ressource dar. Da es mehrere Vertäge geben kann, muss jeder Vertrag durch eine eindeutige URI referenzierbar sein, um die Repräsentation des Vertragszustands abrufen zu können. Der Zustand eines Vertrags kann nicht beliebig zwischen allen Möglichkeiten wechseln. Daher muss durch Hypermediainformationen sichergestellt werden, dass nur die fachlich erlaubten Zustandsübergänge durchgeführt wer-

den.

3.2 Betrachtung der Implementierung

Der in Abschnitt 3.1 beschriebene Geschäftsprozess ist im Rahmen dieser Arbeit mit verschiedenen Open Source Technologien implementiert worden. Da das Fallbeispiel den Anspruch besitzt möglichst realistisch zu sein, sind die Aufgabenbereiche auf mehrere getrennte Dienste verteilt. Schwerpunkt der Implementierung ist die Betrachtung der Kommunikation zwischen den einzelnen Diensten. In Abbildung 2 werden die umgesetzten Kommunikationswege visualisiert. Die Aufteilung der implementierten Dienste ist, wie folgt, vorgenommen:

- *PVP* (Privatkunden Vertriebsportal): Das PVP ist eine Webanwendung für Mitarbeiter in den Filialen des Telekommunikationsunternehmens, um Zugriff auf die internen Dienste zu erhalten und dort Verträge zu verwalten. Die Anwendung ist nach einer 3-Tier Architektur aufgebaut. Im *Frontend* wird das Spring Web MVC-Framework eingesetzt um mittels HTML und AJAX Daten zwischen dem Mitarbeiter und dem Vertriebsportal auszutauschen. An Hand des Frontends wird später untersucht, welche Besonderheiten bei der Verwendung von REST-Diensten durch menschliche Konsumenten berücksichtigt werden müssen.

  Unter der Frontend-Schicht folgt eine Management- bzw. Serviceschicht, in der die Anwendungslogik des Vertriebsportals implementiert ist. Das Backend stellt Zugriffsmöglichkeiten auf Datenquellen wie den Contract and Customer Service (CCS) und den Rating-Service bereit. Zum CCS existiert eine doppelte Anbindung, um zu zeigen, dass SOAP/WSDL-Schnittstellen durch RESTful HTTP-Schnittstellen ersetzt werden können und dass das HATEOAS-Prinzip einen Mehrwert gegenüber RPC-orientierten Clients bietet. Die Kommunikation mit dem Rating-Service erfolgt ausschließlich mitels SOAP, um in Abschnitt 5.3 eine Grundlage für die Anwendung von hybriden Kompositionsansätzen zu besitzen.

- *CCS* (Contract and Customer Service): Dieser Dienst ist zur unternehmensinternen Verwaltung von Vertrags- und Kundendaten vorgesehen. Er ist im Rahmen dieser Arbeit, wie bereits erwähnt, in zwei unterschiedlichen Architekturen implementiert. Die REST-Implementierung verwendet das *JBoss RESTeasy*-Framework, während die SOAP/WSDL-Implementierung das *Apache CXF*[5]-Framework verwendet.

  Aufgabe des Dienstes ist es als eine Abstraktion für Datenbankzugriffe zu agieren und im Fall der REST-Schnittstelle durch Hypermediainformationen die Geschäftsregeln bei den Dienstkonsumenten durchzusetzen. So verändert sich abhängig vom Zustand einer Ressource die Menge der möglichen folgenden Interaktionen mit derselben.

[5] Celtix and XFire sind zwei getrennte Frameworks, die inzwischen zum CXF-Framework zusammengeführt sind. [28]

- *Rating*: Diese Anwendung dient zur Prüfung der Bonität von Personen ähnlich zur Schufa. Innerhalb dieser exemplarischen Implementierung kann leider nicht auf die API der Schufa zugegriffen werden, da diese nicht frei nutzbar sondern nur für Schufa-Vertragspartner zugänglich ist[6].

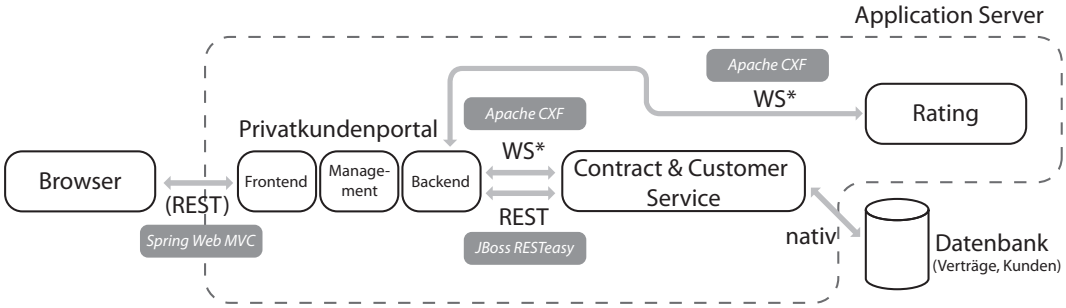

Abbildung 2: Übersicht der Systemkomponenten zum Fallbeispiel

3.2.1 Customer and Contract Service

3.2.1.1 Servlet-Konfiguration

Wie bereits erwähnt, bietet der Contract and Customer Service (CCS) eine SOAP und eine RESTful HTTP API an, um Verträge und Kunden zu verwalten. Die Anfragen an die APIs werden über zwei getrennte Servlets voneinander unterschieden. Die Herausforderung für die Frameworks, ist es auf dieselben Datenbankobjekte zuzugreifen und die Daten synchron zu halten. Der parallele Einsatz der beiden unterschiedlichen Schnittstellen ist jedoch prinzipiell möglich. Dadurch, dass zwei getrennte Implementierungen zur Umsetzung der selben Funktionalität entwickelt und gewartet werden müssen, kommt es bei Arbeiten an den APIs zu einer Verdopplung des Aufwands.

Um die Synchronisation des beiden Servlets umzusetzen, teilen sich beide einen sog. Spring Application Context. Dieser dient als IoC[7]-Container, so dass die Instanziierung und Verbindung zwischen Java-Objekten aus dem Programmcode ausgelagert werden kann. Die zur Implementierung verwendeten Frameworks sind in der Lage sich in diesen Kontext zu integrieren und somit mit geteilten Datenzugriffsobjekten zu arbeiten. Die Synchronisation der Daten erfolgt auf Grund der Wiederverwendung von Datenzugriffsobjekten (Data Access Schicht) bereits in der Dienstanwendung und nicht erst innerhalb der Datenbank.

Aus dem geteilten Application Context ergibt sich ein weiterer Vorteil: Die Klassen der Geschäftsobjekte können wiederverwendet werden, was zu einer einheitlichen Serialisierung der Objekte bzw. Deserialisierung der Repräsentationen unabhängig von der jeweiligen Schnittstelle führt.

[6] http://www.schufa.de/media/teamwebservices/produkteservices/downloads_5/PIB_Anbindungen_XML_-0907.pdf

[7] Inversion of Control

In Abbildung 3 wird der Aufbau der Servlets und des Application Context nochmals verdeutlicht. In Abbildung 22 im Anhang A.5 befindet sich ein Screenshot des der Interaktion mit dem RESTeasy Servlet.

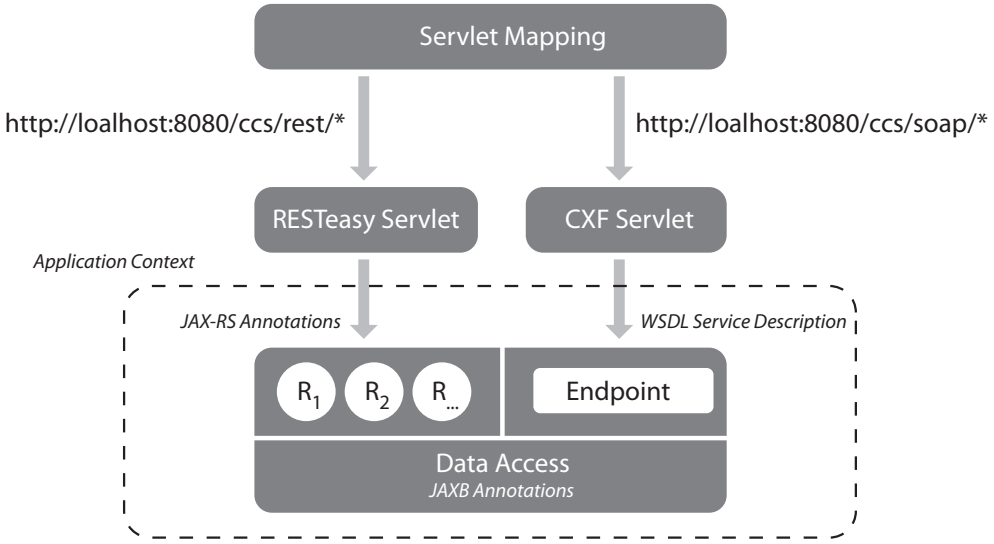

Abbildung 3: Übersicht zur Konfiguration der CCS-Anwendung

3.2.1.2 Abbildung von Anfragen auf Java-Methoden

Das zur Umsetzung der RESTful HTTP-Schnittstelle verwendete RESTeasy-Framework ist konform zur JAX-RS-Spezifikation, die eine Java API zur Implementierung von RESTful Services beschreibt.

Ressourcen werden nach der JAX-RS-Spezifikation als eigene Klassen oder Interfaces umgesetzt. Im Prozess des Fallbeispiels sind Verträge die am häufigsten verwendeten Ressourcen. Daher steuert eine `ContractResource`-Klasse den Zugriff auf alle Verträge. Sie hat die Aufgabe HTTP-Anfragen auf Java-Methoden abzubilden und ggf. ein Antwortobjekt zurückzuliefern. Das RESTeasy-Framework übernimmt die Aufgaben der De-/Serialisierung, so dass der Anwendungsentwickler ausschließlich mit Java-Objekten arbeiten kann.

Damit das RESTeasy-Framework die Anfragen der Clients auf die Ressourcen-Objekte abbilden kann, werden die Resource-Klassen mit URI-Templates annotiert. Diese Templates sind relative Pfade, die Platzhalter für erst zur Laufzeit definierte Informationen enthalten. Um die URI-Templates einzelnen Java-Methoden zuzuordnen, definiert die JAX-RS-Spezifikation die `@Path`-Annotation. Mit dieser können neben Methoden auch Klassen bzw. Interfaces an die URIs gebunden werden. Die Annotation einer Klasse bzw. Interfaces lässt den Wert der Annotation als Prefix für die Annotationen der einzelnen Methoden agieren.

Durch die `@PathParam`-Annotation der JAX-RS-Spezifikation werden die für Platzhalter eingesetzten Werte, als Parameter an die jeweils annotierte Methoden übergeben. Ein

typischer Anwendungsfall ist etwa das Übergeben der Kennung eines Vertrags, der durch einen Client abgerufen wird.

Die Kombination der beiden zuletzt genannten Annotationen dient dazu das Prinzip der eindeutigen URIs für Ressourcen umzusetzen.

Um das Prinzip des uniformen Interfaces umzusetzen, ermöglicht die JAX-RS API die Annotation der Java-Methoden mit den Bezeichnern der HTTP-Methoden. Dadurch lässt sich die Bedeutung einer Anfrage näher einschränken. Zur Auswahl stehen laut JAX-RS-Spezifikation `@GET`, `@POST`, `@PUT` und `@DELETE`. Anfragen mit den HTTP-Methoden HEAD oder OPTIONS werden durch das RESTeasy-Framework automatisch beantwortet.

Das folgende Code-Beispiel zeigt den Einsatz aller genannten Annotationsarten am Beispiel des Abrufs eines Vertrags:

```
@Path("/contracts")
public class ContractResource {

  @GET
  @Path("/{id}")
  public Contract getContractById(@PathParam("id") String id) {
    ...
  }
}
```

Das von einer Methode der Ressourcenklasse zurückgegebene Antwortobjekt kann entweder wie im obigen Codefragment ein beliebiges POJO (Plain Old Java Object) oder aber auch ein `Response`-Objekt aus der JAX-RS API sein. Das `Response`-Objekt bietet dem Entwickler die Möglichkeit Einfluss auf die Statuscodes und Header-Informationen der Antwort zu nehmen. Ein möglicher Anwendungsfall ist z.B. das Verbieten von unerlaubten Aufrufen. Soll ein Vertrag aktiviert werden, dessen Kunde keine positive Bonitätsbeurteilung erhalten hat, so kann mit Hilfe des Response-Objekts der HTTP-Statuscode „Forbidden“ in der Antwort verwendet werden. Das Setzen eines solchen Fehlercodes bietet dem Konsumenten des Dienstes die Möglichkeit zu erkennen, dass seine Anfrage nicht erfüllt werden konnte. Das folgende Codefragment zeigt, wie innerhalb der Methoden der Ressourcenklassen mittels des `Response`-Objekt ein Teil des Prinzips der selbstbeschreibenden Nachrichten umgesetzt wird.

```
public Response activateContract() {
  ...
  if(state.equals("creditworthy")) {
    ...
    return Response.ok(contractActivation).build();
  } else {
    return Response.status(Status.FORBIDDEN).build();
  }
}
```

Man könnte die Kritik vorbringen, dass der Fehlerfall nicht eintreten kann, weil durch Hypermediainformationen die Verknüpfung des Vertrags zu seiner Aktivierung nur dann

enthalten ist, wenn diese auch erlaubt ist. Allerdings sollte aus Sichterheitsgründen innerhalb der Methoden der Ressourcenklassen nochmals geprüft werden, ob die jeweiligen Aktionen überhaupt, vom aktuellen Zustand der Ressource aus betrachtet, erlaubt sind.

Zur Serialisierung bzw. Deserialisierung von POJOs werden JAXB-Annotationen verwendet. Diese sind in erster Linie für XML-Formate ausgelegt, jedoch werden sie ebenfalls intern auch für den Umgang mit dem JSON-Format verwendet. Möglich ist dies, da sich die Abbildungsprozesse zwischen Java und XML bzw. Java und JSON in einigen Aspekten ähneln (siehe [40]).

Statt den Programmcode sämtlicher Klassen und Methoden darzustellen, gibt Tabelle 2 eine Übersicht der gesamten `ContractResource`-Klasse. Die HTTP-Verben/Methoden sind nach der Semantik des HTTP-Standard verwendet [13]. *GET* wird zum Abrufen einer Ressourcenrepräsentation verwendet. Dabei kann der Konsument des Dienstes sicher sein, dass er keine Veränderung am Zustand der abgerufenen oder anderen Ressourcen hervorruft. Mittels *POST* und *PUT* werden Ressourcen erzeugt bzw. aktualisiert. Dennoch dürfen beide nicht als synonym zueinander betrachtet werden. Hierauf wird auf Seite 30 näher eingegangen. Die *DELTE*-Methode wird zum Löschen einer Ressource verwendet. Bei den Verben *GET*, *PUT* und *DELETE* handelt es sich um idempotente Methoden. Der Begriff der Idempotenz wird von Tilkov wie folgt erklärt:

> *„Das ist eigentlich ein mathematischer Begriff, der eine Klasse von Funktionen definiert, die das gleiche Resultat liefern, wenn man sie auf sich selbst anwendet (d. h: bei denen $f(x) = f(f(x))$ ist).“* [45].

Bezogen auf einen HTTP-Request bedeutet dies, dass eine Anfrage mit einer idempotenten Methode beliebig oft wiederholt werden kann ohne dass sich die Antwort ändert. Die POST-Methode hingegen ist nicht idempotent, so dass sie bei nicht wiederholbaren Interaktionen eingesetzt werden sollte.

Eine *POST*-Anfrage soll laut HTTP 1.1 Spezifikation eine untergeordnete Entität zur adressierten Ressource enthalten (siehe [13] S. 53). Dies bedeutet, dass per *POST* an eine URI die Repräsentationen von unterschiedlichen Entitäten gesendet werden können. An eine Listenressource muss demnach nicht automatisch eine Liste gesendet werden. Stattdessen kann auch ein zusätzliches Listenelement per POST an die Listenressource geschickt werden, welches anschließend in die Liste aufgenommen wird. Dieses Verhalten ist vergleichber mit der *späten Bindung* aus der objektorientierten Programmierung, in der sich abhängig vom Typ der verwendeten Parameter erst zur Laufzeit entscheidet, welche Programmlogik ausgeführt wird. In Bezug auf das RESTeasy-Framework ist jedoch festzustellen, dass diese Form der späten Bindung nicht unterstützt wird. Die Kombinationen aus `@Path` und `@GET`/`@POST`/`@PUT`/`DELETE` müssen in den Ressourcenklassen eindeutig sein, während die Signatur der zugeordneten Java-Methode keinen Einfluss auf das Binding von HTTP-Anfragen zu Java-Methoden hat. Der folgende Code-Ausschnitt zeigt ein Beispiel für die **nicht** unterstützte späte Bindung:

```
@Path("/contracts")
```

```
public class ContractResource {

  @POST
  @Path("/{id}")
  public Response updateContract(@PathParam("id") String id, Contract
      contract) {
    ...
  }

  @POST
  @Path("/{id}")
  public Response activateContract(@PathParam("id") String id,
      ContractActivation ca) {
    ...
  }
}
```

Aus diesem Grund muss eine Problemumgehung über eine zusätzliche Subressource erfolgen. In der Implementierung des CCS wird hierzu die Vertragsaktivierung als Subressource zu einem Vertrag eingeführt. Aufgrund des Einsatzes von Hypermedia ist die Verwendung eines zusätzlichen URI-Templates weniger kritisch zu betrachten, da Verknüpfungen in der Repräsentation des Vertrags auf die URI der Vertragsaktivierung verweisen.

3.2.1.3 Ressourcenerzeugung

Da die *POST*- und *PUT*-Methoden des HTTP ähnliche Bedeutungen besitzen, müssen sie näher unterschieden werden. Tilkov [45] stellt heraus, dass die *POST*-Methode verwendet werden soll, wenn der Dienst die URI der zu erstellenden Ressource bestimmt - *PUT* hingegen sollte dann eingesetzt werden, wenn der Client die URI der zu erstellenden Ressource bereits kennt. Aus diesem Grund wird in der `ContractResource`-Klasse die Erstellung eines leeren Vertrags auf eine POST-Anfrage an die Listenressource aller Verträge abgebildet, da die URI des neuen Vertrags nicht bekannt sein kann. Daher enthält der Antwort-Header das `Location`-Feld, welches auf die URI des erzeugten leeren Vertrags verweist. Dieses Vorgehen bietet den Vorteil, dass der Client keine Annahmen darüber treffen muss, wie die URIs von Vertragsressourcen bestimmt werden. Ferner werden Kollisionen mit bereits existierenden Ressourcen vermieden.

Zur Erstellung von Ressourcen weist Tilkov in [45] auf ein Muster hin, damit Ressourcen nicht doppelt erstellt werden. Zunächst soll jedoch das Problem näher beschrieben werden. Ein Anwendungsentwickler könnte verlangen, dass bei der Erstellung eines Vertrags, dessen Daten Teil der *POST*-Anfrage sind. Da *POST* eine nicht-idempotente Methode ist, ist nicht sichergestellt, dass ein solcher Methodenaufruf immer zum gleichen Ergebnis führt. Geht etwa auf Grund von Verbindungsproblemen die Antwort des Dienstes auf eine POST-Anfrage verloren, so muss der Konsument hoffen durch erneutes Stellen der Anfrage eine Antwort zu erhalten. Die Auswirkung wäre jedoch, dass z.B. für den Fall der Erstellung eines Vertrags dieser nun mehrmals existiert. Das von Tilkov vorgestellte Muster löst dieses Problem dadurch, dass die Erstellung einer Ressource in zwei Schrit-

ten erfolgt. Zunächst wird über eine *POST*-Anfrage eine leere Ressource erzeugt. D.h. der Konsument des Dienstes sendet keine Informationen über die zu erstellende Ressource mit. Wie bereits erwähnt, ist in der Antwort die URI der neu angelegten Ressource enthalten. An diese URI werden mittels einer *PUT*-Anfrage die Informationen der neuen Ressource gesendet. Geht eine der Antworten verloren, so ist dies nicht schlimm, da für den Fall der *POST*-Anfragen neu erstellte leere Ressourcen automatisiert aus der Anwendung durch einen „Garbage Collection"-Mechanismus entfernt werden können bzw. im Fall der *PUT*-Anfrage diese auf Grund ihrer Idempotenz ohne Seiteneffekte erneut ausgeführt werden kann.

3.2.1.4 Verknüpfung von Ressourcen

Die Umsetzung des HATOAS-Prinzip von REST wird in der durchgeführten Implementierung durch die Einbettung von Links in die Ressourcenrepräsentationen vorgenommen. RESTeasy stellt hierfür eigene Annotationen bereit. Die `@AddLinks`-Annotation kennzeichnet Java-Methoden, in deren Antwort Verknüpfungen eingesetzt werden sollen. Damit RESTeasy diese Verknüpfungen einfügen kann, sucht das Framework innerhalb der Hierarchie des Antwortobjekts nach einem Attribut des Typs `RESTServiceDiscovery`. Dieses Attribut dient als Container, in den durch das Framework Links injeziert werden.

Verknüpfbare Java-Methoden[8] werden durch die `@LinkResource`-Annotation markiert. Der `Value`-Parameter legt den Objekttyp fest, in den dieser Link prinzipiell eingesetzt werden kann. Ob eine Verknüpfung zu einer solchen Methode hinzugefügt wird, wird durch den `constraint`-Parameter der Annotation gesteuert. Standardmäßig ist dieser Parameter so gesetzt, dass der Link immer eingefügt wird. Durch einen EL[9]-Ausdruck kann gesteuert werden, unter welchen Umständen die Verknüpfung nicht erzeugt werden soll. Innerhalb des EL-Kontexts kann mit der `this`-Variable das Antwortobjekt referenziert werden, in dessen Repräsentation der Link eingesetzt werden soll. Dadurch ist es möglich den Zustand des Objekts auszuwerten.

Für das Fallbeispiel gilt, dass in der Repräsentation eines Vertrags nur dann ein Link auf der Vertragsaktivierung auftauchen darf, wenn der Status des Vertrags auf „creditworthy" gesetzt ist (vgl. Abbildung 1 auf Seite 24). Daher ergibt sich die folgende Konfiguration der `ContractResource`-Klasse:

```
@Path("/contracts")
public class ContractResource {

  @GET
  @Path("/{id}")
  @AddLinks
  public Response getContractById(@PathParam("id") String id) {
    ...
  }
```

[8] Korrekt wäre hier zwar eher die Formulierung, dass Ressourcen miteinander verknüpft werden, jedoch trägt in diesem Fall der Begriff der Methode besser zum Verständnis des RESTeasy-Frameworks bei.

[9] (Java) Expression Language

```
  @PUT
  @Path("/{id}/activation")
  @LinkResource(value=Contract.class, rel="activation", constraint="${
      this.state=='creditworthy'}")
  public Response activateContract(@PathParam("id") String id,
      ContractActivation contractActivation) {
    ...
  }
}
```

Sollte eine Methode mehrere `LinkResource`-Annotationen benötigen, so müssen diese innerhalb der `@LinkResources`-Annotation zusammengefasst werden. Sie dient als Container für `@LinkResoure`-Annotationen, da eine Annotation pro Methode nur maximal einmal verwendet werden kann.

Tilkov [45] weist darauf hin, dass innerhalb eines Client für einen RESTful Service nur eine einzige URI kodiert sein sollte. Über diese URI soll dem Konsumenten der Einstieg in den Dienst ermöglicht werden; alle weiteren Ressourcen sollen dynamisch entdeckt werden. Für diesen Zweck wird in der Implementierung des Fallbeispiels eine `RootResource` -Klasse verwendet, die ein `Services`-Objekt zurückliefert, welches Verknüpfungen auf alle weiteren Ressourcenklassen enthält. Das folgende Codefragment zeigt diesen Zusammenhang in einer reduzierten Form auf.

```
@Path("/")
public class RootResource {
  @GET @AddLinks
  public Response getServiceList(){
    return Response.ok(new Services()).build();
  }
}

@Path("/contracts")
public class ContractResource {
  @GET @AddLinks
  @LinkResources({
    ...
    @LinkResource(value=Services.class, rel="contracts")
  })
  public Contracts getAllContracts(){
    ...
  }
  ...
}
```

3.2.1.5 SOAP-Schnittstelle

Die SOAP-Schnittstelle des Customer and Contract Service wird mittels der JAX-WS API und CXF implementiert. Mittels des CXF wird zur Laufzeit aus der Dienstimplementierung eine WSDL-Beschriebung des Diensts generiert. Ein solches Vorgehen wird auch als

URI-Template	Verb	Beschreibung
/	GET	Liste aller Services
/contracts	GET	Liste aller Verträge abrufen
	POST	leeren Vertrag anlegen
/contracts/{id}	GET	Vertragsinformation abrufen
	PUT	Vertragsdaten aktualisieren
	DELTE	Vertrag löschen
/contracts/{id}/activation	PUT	Vertrag aktivieren und freischalten

Tabelle 2: Übersicht der RESTful HTTP API des Contract and Customer Service

contract last bezeichnet. In einem klassischen Ansatz würde ein *contract first*-Vorgehen angewendet werden, bei dem erst die WSDL-Beschreibung definiert und anschließend die implementierende Dienstklasse bzw. deren Interface generiert wird. Aus Gründen der Einfachheit wurde hierauf jedoch im Fallbeispiel verzichtet.

Dabei zeigt sich, dass im Gegensatz zur Implementierung nach den REST-Prinzipien eine einzige Klasse als Einstiegspunkt für sämtliche Anfragen verwendet wird. Eine Trennung nach der Art der durch eine Anfrage betroffenen Geschäftsobjekte existiert nicht. Prinzipiell ist eine solche Trennung jedoch möglich, da die WSDL die Definition mehrerer sog. Ports erlaubt. Innerhalb eines Ports werden die verfügbaren Bindings einzelnen URIs zugewiesen. Üblicherweise existiert in der Praxis jedoch nur ein einiziges Binding und ein einziger Port.

Eine mit der `@WebService` annotierten Klasse markiert eine Klasse als Dienst. Alle dieser Klasse gehörigen und als `public` deklarierten Methoden werden als Operationen im WSDL-Dokument aufgenommen. Setter- und Getter-Methoden, die nur für dienstinterne Zwecke verwendet werden, aber dennoch als `public` deklariert werden müssen, lassen sich durch eine entsprechend parametrisierte `@WebMethod`-Annotation von der Generierung des WSDL-Dokuments ausschließen.

Im Gegensatz zur Implementierung des Dienstes mittels RESTful HTTP, kann bei der Umsetzung mit SOAP eine beliebig große Menge von Operationen erzeugt werden. Eine Zuweisung zu HTTP-Methoden oder URIs ist nicht notwendig, da jede SOAP-Anfrage durch die POST-Methode des HTTP an eine einzige URI - den Endpunkt - überträgt (siehe [35] S. 811). Somit existieren im Gegensatz zu REST-Architekturen kein uniformes Interface und keine eindeutigen URIs für die Geschäftsobjekte bzw. Ressourcen.

Mittels der `@WebResult`- und `@WebParam`-Annotation wird es ermöglicht die Parameter- und Rückgabe-XML-Knoten zu benennen und den Namespace der übertragenen Repräsentationen zu definieren. Innerhalb der Implementierung der SOAP-Schnittstelle wird im Customer and Contract Service für jede existierende Operation eine `Request`-Klasse und ggf. eine `Response`-Klasse erstellt. Diese dienen als Container für den Inhalt der per SOAP ausgetauschten Objekte. Dies ermöglicht es die Signatur der Methoden des Dienstes möglichst stabil zu halten. Sollte sich ein Dienst dahingehend verändern, dass

eine Operation andere Datenstrukturen als Parameter benötigt, so handelt es sich hierbei nicht um eine Anpassung der Operationssignatur, sondern um eine Anpassung des jeweiligen Request-Datentyps. Innerhalb der Request- und Response-Klassen werden diese mittels JAXB API annotierten Klassen wie in der Implementierung mit RESTeasy verwendet. Dadurch liefert die Serialisierung der Geschäftsobjekte unabhängig von der Dienstschnittstelle dasselbe Ergebnis.

Abweichend zu RESTful HTTP legt SOAP das Datenformat der ausgetauschten Nachrichten auf XML fest. Daher muss in SOAP-Nachrichten keine Metainformation über das Repräsentationsformat ausgetauscht werden. Dies bedeutet, dass ein Konsument des Dienstes keine Möglichkeit hat, die Antwort in seinem bevorzugten Repräsentationsformat anzufordern. Z.B. werden von Browsern oftmals Daten im JSON-Format nachgeladen, um diese ohne zusätzlichen Parsing-Aufwand für den Entwickler mittels JavaScript weiterzuverarbeiten.

Einen weiteren Unterschied zur Implementierung mit RESTeasy stellt das Fehlen von Kontrollflussinformationen dar. Die Webservice-Klasse muss innerhalb jeder als Operation veröffentlichten Methode entscheiden, ob diese Methode auf Grund des aktuellen Anwendungszustands ausgeführt werden darf. Der folgende Codeausschnitt zeigt dies an Hand der Vertragsaktivierung.

```
@WebMethod
public @WebResult(name="activateContractResponse")
    ActivateContractResponse activateContract(@WebParam(name="
    activateContractRequest") ActivateContractRequest request) {
  ActivateContractResponse response = new ActivateContractResponse();

  Contract contract = request.getContract();
  String state = null;
  if (contract != null) {
    state = request.getContract().getState();
  }
  if (state != null && (state.equals("creditworthy") || state.equals("
      alive"))) {
    // if the contract is in the correct state
    long id = new Long(request.getContract().getId());
    contract = contractDAO.activateContract(id);
  }
  response.setContract(contract);

  return response;
}
```

3.2.2 Privatkundenvertriebsportal als Client

Alle der im Folgenden beschriebenen Client-Implementierungen sind so vorgenommen, dass in der Managementschicht des Privatkundenportals mit einer Technologie unabhängigen Schnittstelle gearbeitet werden kann. Hierzu definiert das `IContractService-`

Interface die verfügbaren Java-Methoden. Jede der erstellten Client-Implementierungen verhält sich konform zu diesem Interface, so dass sie beliebig ausgetauscht werden können. Abbildung 4 zeigt eine Übersicht zu den Clients, den Diensten und dem `IContractService` -Interface.

```
public interface IContractService {
  public List<Contract> getAllContracts();
  public Contract createEmptyContract();
  public Contract getContractById(long id);
  public Contract saveContract(Contract contract);
  public boolean deleteContract(Contract contract);
  public Contract activateContract(Contract contract);
}
```

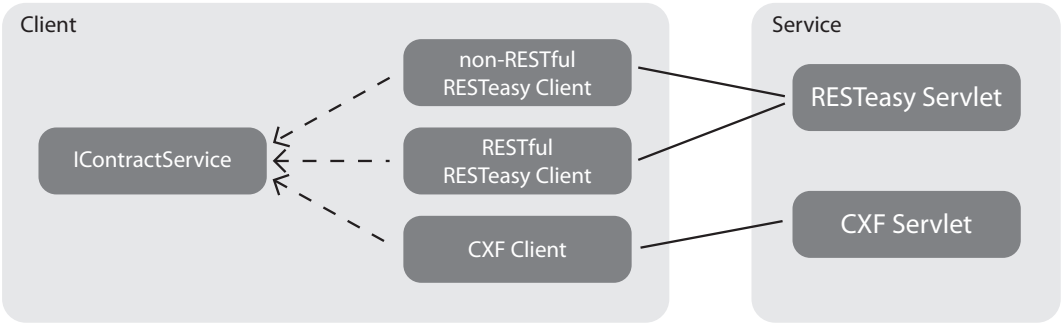

Abbildung 4: Übersicht der umgesetzten Clients und ihre Zuordnung zu den Dienstimplementierungen

3.2.2.1 Non-RESTful RESTeasy

Bei der Implementierung des RESTful HTTP Clients für den Contract and Customer Service wird wie bei der Implementierung des Dienstes das RESTeasy Framework verwendet, da dieses neben der Service API auch eine Client API anbietet. Die Besonderheit der Client API ist, dass der Client als Spiegel des Service aufgebaut wird. Die RESTeasy Dokumentation beschreibt dies auf folgende Weise:

> *„The Resteasy Client Framework is the mirror opposite of the JAX-RS server-side specification. Instead of using JAX-RS annotations to map an incoming request to your RESTFul Web Service method, the client framework builds an HTTP request that it uses to invoke on a remote RESTful Web Service.“* [29].

Laut Dokumentation des Frameworks sollte daher im Client eine Kopie der Dienstschnittstelle vorliegen, um aus dem Aufruf einer Java-Methoden eine HTTP-Anfrage zu generieren. Dabei werden die Annotationen des JAX-RS Standard wieder verwendet, der nur für Dienst- und nicht für Clientimplementierungen konzipiert ist (siehe [17]). Dies hat zur Folge, dass der Client durch die `@Path`- und `@PathParam`-Annotationen mittels vor-

definierter URI-Templates auf den Dienst zugreift. Bei dieser Vorgehensweise ergibt sich folgendes Java Interface:

```
@Path("/contracts")
@Consumes("application/xml")
public interface ContractResource {

  @GET
  public Collection<Contract> getAllContracts();

  @POST
  public ClientResponse<Contract> createContract();

  @GET @Path("/{id}")
  public ClientResponse<Contract> getContractById(@PathParam("id") String
       id);

  @PUT @Path("/{id}")
  public ClientResponse<Contract> updateContract(Contract contract,
      @PathParam("id") String id);

  @DELETE @Path("/{id}")
  public Response deleteContract(@PathParam("id") String id);

  @PUT @Path("/{id}/activation")
  public ClientResponse<ContractActivation> activateContract(@PathParam("
      id") String id, ContractActivation contractActivation);
}
```

Die `ClientResponse`-Klasse stammt aus dem RESTeasy-Framework und abstrahiert die Antwort des Dienstes. So kann mittels der `ClientResponse`-Klasse durch Verwendung von Generics der Java-Typ des in der Nachricht kodierten Objekts spezifiziert werden. Dies ist hilfreich, um bei der Deserialisierung der Nachricht zu wissen, welcher POJO-Klasse das repräsentierte Objekt entspricht. Darüber hinaus enthält das `ClientResponse` -Objekt Informationen über den HTTP-Statuscode und den HTTP-Header.

Damit die im Interface deklarierten Methoden aufrufbar werden, wird durch Verwendung der `ClientRequestFactory` des RESTeasy-Frameworks eine Proxyklasse generiert und instanziiert, die dieses Interface implementiert. Die Proxyklasse hat die Aufgabe, die tatsächlichen HTTP-Anfragen zu erzeugen und die erhaltenen Antworten zu verarbeiten. Damit die relativen Pfadangaben aus den `@Path`-Annotationen in absolute Adressen umgewandelt werden, wird bei der Erzeugung der `ClientRequestFactory` eine Basis-URI definiert, die bei der Generierung des Proxy weiterverwendet wird.

Das folgende Codefragment zeigt die beschriebene Erzeugung des Client-Proxy.

```
httpClient = new HttpClient(new MultiThreadedHttpConnectionManager());
executor = new ApacheHttpClientExecutor(httpClient);
uri = new URI(BASE_URI);
clientRequestFactory = new ClientRequestFactory(executor, uri);
```

```
contractResource = clientRequestFactory.createProxy(ContractResource.
    class);
```

Durch das Vorgehen der Annotation eines Ressourcen-Interface mittels JAX-RS und der anschließenden Proxygenerierung wird es dem Client ermöglicht zu jedem Zeitpunkt jede beliebige Interaktion mit einer Ressource durchzuführen. Damit verzichtet der Client auf die Synchronisation des eigenen Zustands mit dem Zustand der im Dienst existierenden Ressourcen. Das HATEOAS-Prinzip wird nicht beachtet, da die vom Dienst bereitgestellten Hypermediainformationen bei der Erzeugung einer HTTP-Anfrage nicht verwendet werden.

Eines der Kernkonzepte von REST ist, dass sich der Client durch den Zustand der Ressourcen steuern lässt. Dies erfordert jedoch, dass der Client den Ressourcenzustand kennt, bevor neue Interaktionen erfolgen. Daher sollte der Client die Links aus den vorangehenden Nachrichten verwenden und auf die Kodierung von URI-Templates verzichten.

Paradoxerweise werden die Links aus den vom Dienst erhaltenen Nachrichten bei der Deserialisierung in ein Attribut vom Typ `RESTServiceDiscovery` geschrieben, falls die jeweilige Java Klasse des Geschäftsobjekts dieses Attribut besitzt (vergleiche Abs. 3.2.1.4 auf S. 31). RESTeasy ist zwar in der Lage Links zu generieren, in die Repräsentation einzufügen und zu deserialisieren, jedoch kann es diese Informationen in der eigenen Client API nicht automatisch nutzen.

Dieser Mangel an der Vorgehensweise zur Entwicklung von Clients mit dem RESTeasy-Framework ist in einer alternativen Implementierung kompensiert, die im nächsten Abschnitt näher betrachtet wird.

3.2.2.2 RESTful RESTeasy

Damit der Client für den Contract and Customer Service REST-konform ist, muss der Zustand des Clients durch die Verknüpfungen in den Ressourcenrepräsentationen gesteuert werden. Hierfür müssen zunächst die im Client kodierten URI-Templates entfernt und die URI's für die HTTP-Anfragen aus den Links der vorangehenden Kommunikation mit dem Dienstes entnommen werden. Das bedeutet, dass das `ContractResource`-Interface reduziert wird, indem alle `@Path`-Annotationen und alle mit `@PathParam` annotierten Parameter entfernt werden. Die Annotationen für die HTTP-Methoden bleiben im Java Interface, da sie eine feste Semantik besitzen, die von Dienst und Client geteilt wird.

In dieser Form würde ein generierter Proxy seine Anfragen immer an die Basis-URI schickt. Da sich die zu verwendenden URIs jedoch voneinander unterscheiden, muss der Proxy für jede Anfrage mit der für diese Anfrage passenden URI neu generiert werden. Ein nachträgliches Ändern der URI innerhalb eines Clientproxy ist in der vorliegenden Implementierung nicht gelungen. Die bei der Generierung zu verwendende URI muss aus den Links der vorangehenden Antworten des Dienstes entnommen werden. Daher wird es notwendig, dass der Client einen eigenen Zustand verwaltet, in dem er die Links zu den aktuell verfügbaren Interaktionen der jeweiligen Ressourcen speichert.

Für eine Methode zum Dienstaufruf ergibt sich der folgende Pseudocode. Der vollstän-

dige Java-Code ist im Anhang dieser Arbeit zu finden.

```
public Contract someInteraction(Contract contract) {
  uri = getURIfor("interaction", contract);
  if(uri == null) { return contract; }
  recreateProxyWithURI(uri);
  response = contractResource.interact(contract);
  if(response.getStatus() != ok) { throw new Exception() }
  else {
    clientState = response.getContract().getLinks();
    return response.getContract();
  }
}
```

3.2.2.3 SOAP

Zur Implementierung eines SOAP-Clients wird das generierte WSDL-Dokument beim Dienst heruntergeladen. Mittels des „CXF Codegen Plugin“ lässt sich durch Maven in der generate-sources Phase des Build-Prozesses aus der Dienstbeschreibung der Client für das SOAP-Interface des Contract and Customer Service generieren.

Der so erstellte Webservice-Client kann ähnlich zu ersten RESTeasy-Implementierung zu jedem Zeitpunkt jede Operation aufrufen. Ob diese Operationen fachlich erlaubt sind, muss der Client selbst entscheiden.

Innerhalb der Konfiguration des Spring Application Context (siehe Abs. 3.2.1.1 auf S. 26) wird die Instanziierung des Client durch eine JAX-WS Proxy Factory deklariert. Ähnlich zur Generierung des RESTeasy Resource Proxy wird der Factory das Service-Interface übergeben, das dann vom generierten Proxy implementiert wird. Außerdem kann die URI des Endpunkts manuell angegeben werden, falls dieser von der Adresse des Port im WSDL-Dokument abweicht.

3.2.3 Privatkundenportal als Service

Das Privatkundenportal arbeitet nicht nur als Client des Contract and Customer Service, sondern auch als Dienst, der einem Endanwender Zugriff auf interne Systeme des Telekommunikationsunternehmens verschafft. Bei der Umsetzung des Dienstes wird das Spring Web MVC-Framework verwendet. Dieses besitzt einige Ähnlichkeiten zum RESTeasy-Framework: Beide Frameworks ermöglichen mittels Annotationen eine Abbildung von HTTP-Anfragen auf Java-Methoden. Daher stellt ein Spring Web MVC Controller das Äquivalent zur Ressourcenklasse im RESTeasy-Framework dar. An dieser Stelle soll jedoch weniger auf den Programmcode eingegangen werden, sondern ein stärkerer Fokus auf das Design der API des Privatkundenvertriebsportals gerichtet werden. In Tabelle 3 ist diese API zusammengefasst. Bei der Implementierung des Privatenkundenvertriebsportals wird wissentlich gegen Prinzipien von REST verstoßen, um auf häufig gemachte Fehler bei der Implementierung einer REST-Architektur aber, auch auf Einschränkungen durch die Berücksichtigung von Browser-Clients hinzuweisen. Diese Verstöße sollen nun näher be-

URI-Template	Verb	Beschreibung
/contract	GET	Liste aller Verträge abrufen
	POST	Vertragsdaten aktualisieren und ggf. Aktivierung des Vertrags
/contract/new	GET	leeren Vertrag anlegen
/contract/{id}	GET	Vertragsinformation abrufen
/contract/{id}/print	GET	Vertragsinformation als PDF abrufen
/contract/{id}/directDebitMandate	GET	Einzugsermächtigung als PDF abrufen
/contract/{id}/creditworthiness	GET	Bonitätsinformationen abrufen und im Vertrag speichern

Tabelle 3: API des ContractController aus dem Privatkundenvertriebsportal

trachtet werden. In den Abbildungen 20 und 21 im Anhang A.5 befinden sich Screenshots der Benutzeroberfläche des Privatenkundenvertriebsportals.

3.2.3.1 Verletzung der Idempotenz

Das Konzept der Idempotenz für die *GET*-Methode wird beim Abruf der Bontitätsinformationen verletzt, da diese Informationen nicht nur abgerufen, sondern auch in der Vertragsressource gespeichert werden, ohne dass es für den Dienstkonsumenten sichtbar ist. Dies kann dazu führen, dass die Ressource ihren Zustand ändert, obwohl eine sichere Methode aufgerufen worden ist. Das folgende Codefragment zeigt die Missachtung sowohl der Idempotenz, als auch der unverändernden Wirkung der *GET*-Methode.

```
@RequestMapping(value="/{id}/creditworthiness", method=RequestMethod.GET)
public ModelAndView getCreditworthiness(@PathVariable long id){
  ModelAndView mav = new ModelAndView();
  Contract contract = contractService.getContractById(id);
    if (accountingService.isCustomerCreditworthy(contract.getCustomer()))
        {
      mav.addObject("creditworthiness", Boolean.TRUE);
      contract.setState(Contract.STATE_CUSTOMER_CREDITWORTHY);
    } else {
      mav.addObject("creditworthiness", Boolean.FALSE);
      contract.setState(Contract.STATE_CUSTOMER_NOT_CREDITWORTHY);
    }
    contractService.saveContract(contract);
    return mav;
  }
}
```

3.2.3.2 Verletzung der Semantik des uniformen Interfaces

Das Anlegen eines neuen Vertrags mittels einer GET-Anfrage könnte im Fall des Privatkundenportals als Missachtung der unverändernden Eigenschaft der GET-Methode betrachtet werden. Dies stellt jedoch kein Problem dar, da für jeden erneuten Aufruf dieselbe Antwort gesendet wird. Ob in einem System nicht benötigte Vertragsinstanzen entstehen, ist nicht von Bedeutung, da die Unveränderbarkeit lediglich die Bedeutung hat, dass *„der Nutzer mit*

dem Aufruf und dem damit verbundenen Lesen von Daten keine Verpflichtungen eingeht“ (siehe Tilkov [45] S. 49). Der Nebeneffekt der ggf. unnötig angelegten Ressourcen lässt sich dadurch vermeiden, dass nicht bereits beim Abruf, sondern erst nach dem Abschicken eines Formulars die tatsächlichen Instanzen erzeugt werden.

```
@RequestMapping(value="/new", method=RequestMethod.GET)
public ModelAndView createEmptyContract() {
  ModelAndView mav = createContractView();
  Contract contract = contractService.createEmptyContract();
  mav.addObject("contract", contract);
  return mav;
}
```

3.2.3.3 Missachtung der Selbstbeschreibung

Ein Aspekt der Selbstbeschreibung von Nachrichten ist die Aushandlung der Repräsentationsformate (*Content-Type Negotiation*). Im Accept-Header der Anfrage listet der Client seine bevorzugten Formate auf. Der Dienst muss entscheiden, ob er eines dieser Format unterstützt. Sollte die Verwendung keines der bevorzugten Formate möglich sein, so muss der Dienst mit einem entsprechenden Status antworten, der den Client darüber informiert, dass seine Anfrage nicht bearbeitet werden kann. Wird in der aktuellen Implementierung die Liste der Verträge im JSON- oder XML-Format angefordert, so antwortet der Dienst mit der HTML-Repräsentation anstatt den Statuscode 406 (NOT ACCEPTABLE) zu senden. In den folgenden Codezeilen ist sowohl der Header der Anfrage für eine JSON-Repräsentation aller Verträge als auch der Header der Antwort aufgelistet, um die Missachtung des Accept-Headers zu zeigen.

```
curl -v -H "Accept: application/json" http://localhost:8080/pvp/contract
  GET /pvp/contract HTTP/1.1
  Host: localhost:8080
  Accept: application/json

  HTTP/1.1 200 OK
  Server: Apache-Coyote/1.1
  Content-Type: text/html;charset=UTF-8
  Content-Length: 2331
```

Zur Umgehung dieser Problematik können zusätzliche Subressourcen eingeführt werden, die jedoch nur für bestimmte Formate ausgelegt sind. Das Privatkundenvertriebsportal nutzt eine gesonderte URI für die Darstellung eines Vertrags als PDF-Dokument: `/contract/id/print` . Zwar wird dadurch der Vertrag explizit als PDF-Dokument referenzierbar, jedoch sollte auch ohne die künstliche Subressource (`print`) eine PDF-Reprä-sentation mit dem Dienst aushandelbar sein.

3.2.3.4 Einschränkung des uniformen Interface

Durch die Tatsache, dass das Privatkundenvertriebsprotal primär als browserbasierte Anwendung genutzt wird, kommt als Repräsentationsformat HTML zum Einsatz. Üblicherweise werden HTML-Formulare eingesetzt, um Ressourcen zu erstellen oder zu bearbeiten.

Da das HTML-Format für Formulare jedoch nur die *GET*- und *POST*-Methode des HTTP unterstützt, können sowohl die *PUT*- als auch *DELETE*-Methode nicht verwendet werden. Das uniforme Interface des HTTP wird dadurch eingeschränkt, so dass jede Ressourcen verändernde Funktionalität durch eine POST-Methode abgedeckt werden muss. Auf diese Problematik macht Tilkov (siehe [45] S. 55) aufmerksam.

Tilkov nennt die Verwendung eines versteckten Formularfelds zur Codierung der tatsächlichen HTTP-Methode als eine mögliche Umgehung des Problems. Dabei wird jedoch vorausgesetzt, dass das serverseitige Framework diese versteckten Informationen ausliest und die eingehende Anfrage entsprechend uminterpretiert.

Eine zusätzliche Alternative stellt die Verwendung von AJAX (Asynchronous JavaScript And XML) dar, da durch das `XMLHttpRequest`-Objekt jede HTTP-Methode verwendet werden kann. Dieses Vorgehen ist im in der Implementierung des Privatkundenvertriebsportals für das Löschen von Vertragsressourcen gewählt worden. Um den dafür nötigen JavaScript-Code Browser-übergreifend lauffähig zu gestalten, ist die jQuery-Bibliothek verwendet worden. Das folgende Code-Beispiel zeigt, wie die *DELETE*-Methode mittels der `ajax`-Funktion von jQuery verwendet werden kann.

```
<span class="fakelink" onclick="$.ajax({
    url: '/pvp/contract/${contract.id}',
    context: this,
    type: 'DELETE',
    complete: function(){
      $(this).parent().parent().parent().remove();
    }
  });">löschen</span>
```

Tilkov weist in [45] darauf hin, dass mit HTML5 sowohl die *PUT*- als auch *DELETE*-Methode in Formularen verwendet werden können. Inzwischen sieht die aktuelle Empfehlung des W3C jedoch keine derartige Unterstützung mehr vor[10].

3.3 Beurteilung der Implementierung

Für einen Dienstanbieter ist der parallele Betrieb einer RESTful HTTP und einer SOAP-Schnittstelle kein großes technisches Problem - vorausgesetzt, dass der Dienst die Logik der Anwendungsdomäne nicht mit der Kommunikationslogik vermischt. Dies wird bei der Implementierung des Contract and Customer Service deutlich, da hier zwei unterschiedliche Servlets Zugriff auf dieselbe Anwendungslogik bieten.

Daher ist es denkbar, dass ein Architekturwechsel für eine bereits existierende Anwendung durch einen Parallelbetrieb von unterschiedlichen Schnittstellen prinzipiell möglich ist. Ein solches Vorgehen bietet den Vorteil, dass nicht alle Dienstkonsumenten zur gleichen Zeit auf die neue Schnittstelle umgestellt werden müssen. Langfristig betrachtet bedeutet das Anbieten von zwei Schnittstellen jedoch einen größeren Wartungsaufwand. Schnittstellenänderungen an mehreren Orten synchron durchzuführen, bedeutet Mehraufwand.

[10] siehe http://www.w3.org/TR/2010/WD-html5-diff-20101019/ Abschnitt 5.1.

Die Implementierung von Architekturen auf der Basis von SOAP und WSDL in Java stellt durch die aktuell verfügbaren Softwarewerkzeuge keine große Hürde dar. Die Substituierbarkeit mit RESTful HTTP im konkreten Szenario wird bei der Betrachtung des Client (Privatkundenvertriebsportal) deutlich: Sowohl der SOAP-Client als auch der RESTful HTTP-Client implementieren das `IContractService`-Interface, welches alle Methoden definiert, die für das konkrete Fallbeispiel im Umgang mit Vertragsobjekten notwendig sind. Gleichzeitig wird deutlich, dass das HATEOAS-Prinzip kein Gegenstück in der Implementierung mittels SOAP und WSDL besitzt. Durch das `IContractService`-Interface wird der Umgang mit Links vor dem Anwendungsentwickler verborgen, so dass der Zustand einer Ressource manuell beurteilt werden muss; d.h. wenn der Entwickler die Existenz einer Schaltfläche im Frontend vom Vorhandensein eines Links abhängig machen will, so kann er hierzu nicht das `IContractService`-Interface verwenden, sondern muss auf die Abstraktion verzichten und direkt mit dem REST-Client arbeiten. Auf diese Weise können Steuerungselemente im Frontend von der Existenz von Verknüpfungen abhängig gemacht werden.

Gegenüber einer RPC-orientierten Implementierung, wie z.B. mittels SOAP, bietet dieses Verfahren den Vorteil, dass im Client kein Wissen kodiert werden muss, für welche Ressourcenzustände welche Interaktionen mit einer Ressource möglich sind. Zwar könnten nicht erlaubte Anfragen von einem SOAP-Service mit einer Fehlermeldung beantwortet werden, jedoch würde es in einer durch Hypermedia gesteuerten Anwendung gar nicht erst zu dieser Anfrage kommen. Darüber hinaus kann ein RESTful Service die Regeln frei ändern, wann welche Verknüpfungen vorhanden sind. Für die möglichen RESTful Clients bedeutet dies, dass sie nicht angepasst werden müssen, da sie das HATEOAS-Prinzip beachten.

Individuelle Vorteile von RESTful HTTP wie Content-Type Negotiation werden zwar implementiert, jedoch durch den Client nur eingeschränkt genutzt. Dies wird dadurch deutlich, dass das `ContractResource`-Interface des RESTeasy Clients für jede Methode XML als Repräsentationsformat anfordert.

Es bedeutet einen Mehraufwand für den Anwendungsentwickler bei der Verwendung von RESTeasy den Client RESTful zu gestalten. Alternative Frameworks können diesen Mehraufwand reduzieren. Auf einige Alternativen wird in Kapitel 4.3 näher eingegangen.

Die Implementierung des Privatkundenvertriebsportals zeigt, dass ein Dienst durch die Verwendung eines spezifischen Frameworks oder des HTTP nicht automatisch RESTful umgesetzt wird. Das HTTP zur Kommunikation zwischen Systemkomponenten einzusetzen, reicht nicht aus. Ein Anwendungsentwickler hat die Möglichkeit gegen die Semantik der Methoden des uniformen Interfaces zu verstoßen, die Metadaten der ausgetauschten Nachrichten zu ignorieren oder durch fehlendes Wissen notwendige zentrale Prinzipien wie z.B. Hypermedia zu vernachlässigen.

Um einen Client RESTful zu gestalten, muss der Anwendungsentwickler die REST-Prinzipien achten und besonders im Fall von HATEOAS auch nutzen.

4 Umsetzung von Hypermedia

Im vorangegangen Kapitel ist u.a. die Umsetzung des Hypermedia-Constraints als eine Hürde bei der Entwicklung von RESTful Anwendungen herausgearbeitet worden. Aus diesem Grund werden nun sowohl das HTTP als Anwendungsprotokoll, JAX-RS als Spezifikation für RESTful Webservice Implementierungen und verschiedene Frameworks dahingehend betrachtet, was sie zur Umsetzung des Hypermedia-Constraints beitragen können. Die Auswahl der Frameworks beschränkt sich ausschließlich auf solche, die zur Verwendung in Java Programmen vorgesehen sind. Gründe hierfür sind, dass ich in der Java-Programmierung das größte Hintergrundwissen besitze und Java im Unternehmensumfeld die am häufigsten eingesetzte Programmiersprache ist[11].

4.1 HTTP

Wenn das HTTP betrachtet wird, muss auf dessen unterschiedliche Versionen geachtet werden. Daher ist es bei der Untersuchung des HTTP auf die Umsetzung des HATEOAS-Prinzips notwendig, eine Unterscheidung zwischen den Versionen 1.0 und 1.1 vorzunehmen. In Version 1.0 wird das HTTP im RFC 1945 wie folgt beschrieben:

> *„The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems.“* [6]

Damit ist das HTTP prinzipiell dazu geeignet bei der Umsetzung einer REST-Architektur verwendet zu werden. Da Fielding REST ohne einen expliziten Zusammenhang zum HTTP entwickelt hat, kann das HTTP nicht als alleinstehende Umsetzung der REST-Prinzipien betrachtet werden. Daher werden nun mehrere unterschiedliche Aspekte betrachtet, die an der Umsetzung des HATEOAS-Prinzip mitwirken.

4.1.1 Statuscodes

Diese Statuscodes sind Teil der Antwort auf eine HTTP-Anfrage und stellen die Grundlage zur Interpretation der Antwort dar. Es existieren fünf Klassen von Statuscodes, welche in den Spezifikationen [6], [13] des HTTP 1.0 und 1.1 wie folgt definiert sind:

- **1****: ungenutzte Platzhalterklasse für generische Informationen
- **2****: Klasse für Anfragen, die erfolgreich empfangen, verstanden und bearbeitet werden können
- **3****: Klasse für Anfragen, die zur erfolgreichen Bearbeitung weitere Aktionen erfordern

[11] siehe http://www.heise.de/developer/meldung/Umfrage-Java-ist-verbreitetste-Programmiersprache-bei-IT-Dienstleistern-1198249.html

- **4****: Klasse für ungültige oder nicht erfüllbare Anfragen des Clients
- **5****: Klasse für gültige Anfragen, die durch den Server jedoch nicht bearbeitet werden können

Innerhalb dieser Klassen befinden sich die Statuscodes, die dem Konsumenten des Dienstes signalisieren, dass bestimmte Hypermediainformationen innerhalb der Antwort vorhanden sind. Wie in Abschnitt 3.2.1.3 auf Seite 30 erwähnt, können bei der Erstellung einer neuen Ressource sowohl die Methoden *POST* als auch *PUT* verwendet werden. Da bei der *POST*-Methode, die URI der neuen Ressource durch den Dienst festgelegt wird, muss diese URI zum Client gesendet werden. Aus diesem Grund schreibt das HTTP für den Statuscode `201 CREATED` vor, dass die URI zu einer neu angelegten Ressource im Entity-Bereich der Antwort mit diesem Statuscode enthalten sein muss. Durch diese Hypermedia-Information wird es dem Client ermöglicht, dass er beim Anlegen neuer Ressourcen nicht nach der neuen Ressource suchen oder gar die URI mittels eines Templates selber bestimmen muss.

Ein ähnlicher Anwendungsfall für Hypermedia ist die Verlagerung von Ressourcen. Es kann vorkommen, dass ein Dienst unter einer anderen Adresse angesprochen werden muss (z.B. über einen Proxy oder einen Filter) oder dass ein Dienst durch eine neue Version ersetzt wird. Beide Szenarien haben zur Folge, dass die URIs von existierenden Ressourcen nicht mehr verwendet werden sollen. Nicht immer können alle Konsumenten über derartige Änderungen durch den Dienstanbieter informiert werden. Daher sollten Anfragen, die an eine nicht mehr zu verwendende URI gesendet werden, mit dem Statuscode `301 MOVED PERMANENTLY` beantwortet werden. Die Spezifikationen des HTTP legen für diesen Statuscode fest, dass die zu benutzende URI im `Location`-Feld des Antwort-Headers enthalten sein muss. Auf diese Weise, kann ein Client seine Anfrage automatisiert umlenken, indem er dieser Verknüpfung folgt.

4.1.2 Header Elemente

Statuscodes legen für einige Fälle fest, wann an welcher Stelle einer Antwort durch den Client nach Hypermediainformationen gesucht werden muss. Neben dem Enitity-Bereich einer HTTP-Nachricht kann auch der Header-Bereich abgesehen vom bereits erwähnten `Location`-Feld Hypermediainformationen enhalten.

Die Version 1.0 des HTTP definiert einen optionalen Link-Header, der für jede Ressource, die im Entity-Bereich repräsentiert wird, Zusammenhänge zu anderen Ressourcen aufzeigen kann. Der Vorteil dieses Headers ist, dass er unabhängig vom Datenformat der Repräsentation ist. Somit wird es ermöglicht, dass auch Formate, die nicht für den Umgang mit Hypermediainformationen entwickelt wurden (z.B. JPEG, ZIP etc.), mit Kontrollflussinformationen angereichert werden können. Mit der Nachfolgeversion 1.1 des HTTP ist dieser Header jedoch nicht mehr aufgelistet (siehe [13]). Er taucht in der Spezifikation allerdings auch nicht in den Unterschieden zur Vorgängerversion auf. Daher besitzt der

Link-Header laut Kelly et al. [20] einen undefinierten Status. Seine Verwendung kann nicht explizit empfohlen, aber auch nicht explizit untersagt werden.

4.1.3 Methoden

Das HTTP 1.0 stellt mit *LINK* und *UNLINK* Methoden bereit, die explizit für die Verwaltung von Verknüpfungen zwischen Ressourcen vorgesehen sind. Sie sind jedoch mit Version 1.1 wieder entfernt worden, weil sie bis zur Entwicklung des HTTP 1.1 RFC nur selten implementiert wurden (siehe S. 174 in [13]). Zweck dieser Methoden ist es gewesen, einer Ressource durch zusätzliche Anfragen Verknüpfungen hinzuzufügen bzw. wieder zu entfernen.

Im Gegenzug zu den entfernten Methoden ist die *OPTIONS*-Methode im HTTP 1.1 als Neuerung hinzugekommen. Sie dient dazu Kommunikationsmöglichkeiten mit der aktuell angeforderten Ressource zu entdecken. Sie eignet sich dafür anzufragen, welche Methoden auf die aktuell adressierte Ressource angewandt werden können. Die verfügbaren Methoden werden im `Allow`-Header der Antwort aufgelistet (vgl. Abs. 3.2.1.2). Daher stellt das HTTP eine Möglichkeit bereit das uniforme Interface in die Umsetzung des HATEOAS-Prinzips mit aufzunehmen.

4.2 JAX-RS

Laut Tilkov [45] dient die JAX-RS API allgemein zur Bindung von HTTP-Anfragen an die Methoden von Ressourcenklassen. Dabei wird jede Art von Ressource durch eine eigene Klasse implementiert (vgl. Abschnitt 3.2.1.2). Das Ziel von JAX-RS ist es, alle Konstrukte von HTTP für Anwendungsentwickler verfügbar zu machen. Ein zusätzliche Abstraktion des HTTP wird bewusst nicht vorgenommen (siehe [45]).

4.2.1 Version 1.1

Aktuell ist JAX-RS in Version 1.1 (siehe JSR 311) verfügbar, wobei die erste Veröffentlichung von JAX-RS im Oktober 2008 erfolgt ist. In der Einleitung der Spezifikation [17] wird herausgestellt, dass eine Client API für RESTful Services nicht Bestandteil der Spezifikation ist:

> *„The specification will not define client-side APIs. Other specifications are expected to provide such functionality.“* [17].

Auf Grund dieser Beschränkung auf die Dienstseite, kann JAX-RS 1.1 lediglich auf die Bereitstellung von Hypermediainformationen untersucht werden. Die Konsumierung dieser Informationen wäre Bestandteil der fehlenden Client API.

In Abschnitt 4.1 sind die unter anderem Methoden des HTTP herausgestellt worden,die für die Umsetzung des Hypermedia-Constraints eingesetzt werden können. Aus diesem

Grund soll nun die *OPTIONS*-Methode in JAX-RS näher betrachtet werden. Die JAX-RS-Spezifikation legt fest, dass Anfragen mit dieser Methode automatisch durch die JAX-RS-Implementation (d.h. die jeweiligen Frameworks) beantwortet werden müssen, indem die JAX-RS-Annotationen der jeweiligen Ressourcenklasse ausgewertet werden. Anschließend müssen im Allow-Header die zur Verfügung stehenden HTTP-Methoden aufgelistet werden. Durch dieses Vorgehen besitzt eine Ressource einen statischen Zustand, da keine Einschränkungen für die Verwendung der einzelnen Methoden vorhanden sind; d.h., dass zu jedem Zeitpunkt jede Methode aufgerufen werden kann. Durch den Entwickler kann jedoch eine manuelle Implementierung an die *OPTIONS*-Methode in der Ressourcenklasse gebunden werden, falls der Ressourcenzustand Einfluss auf die Verfügbarkeit einzelner Methoden besitzt. Somit wird durch JAX-RS das HATEOAS-Prinzip dahingehend unterstützt, dass die Verwendung der HTTP-Methoden zur Laufzeit gesteuert wird.

Abhängig vom Statuscode müssen in den Header der Antwort Hypermediainformationen eingefügt werden. Hierzu definiert JAX-RS die `Response`-Klasse, welche mit Hilfe eines `ResponseBuilder` instanziiert wird. Der `ResponseBuilder` unterstützt den Anwendungsentwickler hierbei, indem er z.B. die Erstellung des Location-Header übernimmt, falls die Antwort den Status `CREATED` besitzt (vgl. Abs. 3.2.1.2 auf S. 28). Die URI der erstellten Ressource muss dabei jedoch manuell vom Entwickler als Parameter an den `ResponseBuilder` übergeben werden. Die erstellte `Response`-Instanz kann darüber hinaus um Header-Informationen wie etwa Link-Header erweitert werden. Dies muss jedoch innerhalb jeder Methode der Ressourcenklassen manuell erfolgen. Link-Header sind durch JAX-RS nicht dezidiert unterstützt, jedoch auch nicht explizit unterbunden.

Ähnlich verhält es sich mit Links innerhalb der Repräsentationen. Diese müssen ebenfalls manuell im Programmcode erzeugt werden. Ein einheitliches Vorgehen wird durch die JAX-RS-Spezifikation nicht definiert. Eine Möglichkeit ist es, mittels eigener Link-Klassen und zusätzlichen Attributen innerhalb der POJO-Klassen Hypermediainformationen in die Objekte der Anwendung aufzunehmen. Die Konstruktion der URIs muss dabei vom Entwickler manuell durchgeführt werden, da sie nicht durch die JAX-RS API abgedeckt wird. Grund hierfür ist, dass JAX-RS lediglich für die Bindung von HTTP-Anfragen an Methoden der Ressourcenklassen entworfen ist. Aus diesem Grund bieten JAX-RS implementierende Frameworks, wie z.B. RESTeasy, hierfür zusätzliche Funktionalitäten an. Darauf wird im nächsten Abschnitt näher eingegangen.

Zusammenfassend lässt sich festhalten, dass JAX-RS nur eine geringe Unterstützung von Hypermedia für Anwendungsentwickler in Form des Allow- und Location-Header bietet. Dies wird sogar von Marc Hadley, einem der leitenden Autoren der Spezifikation, in einem persönlichen Fazit nach der Veröffentlichung von JAX-RS gegenüber Tilkov bestätigt:

> *„If I had to identify a weakness [in the JAX-RS API] it would have to be limited support for hypermedia as the engine of state - whilst we provide good support for extracting information from request URIs and building URIs to resources,*

its still very much left to the developer to use hypermedia in representations appropriately.“ [44].

4.2.2 Version 2.0

Zum Zeitpunkt der Entstehung dieser Arbeit ist die Spezifikation zu JAX-RS 2.0 noch in der Enwicklungsphase. Daher lassen sich nur geplante Eigenschaften der kommenden Version nennen. Die größte Neuerung wird die Spezifikation einer Client API darstellen (siehe [27]). Diese ist von den Entwicklern der Java Community in den Vorgängerversionen von JAX-RS vermisst worden (siehe [44]). Laut Beschreibung des JSR 339 [27] ist eine zweistufige API geplant, wobei den Entwicklern eine low-level und eine high-level Variante zur Auswahl angeboten werden.

JAX-RS 2.0 wird darüber hinaus einen Weg definieren, wie Links in den Antwort-Header oder die Repräsentation einer Ressource eingebettet bzw. aus diesen verarbeitet werden. Damit wird es eine Framework übergreifende einheitliche Lösung zur Handhabung von Hypermediainformationen geben.

Eine detailiertere Bewertung der Unterstützung von Hypermedia durch JAX-RS 2.0 wird voraussichtlich im vierten Quartal 2011 möglich werden, da die Entwickler dann ein öffentliches Review der Spezifikation anstreben. Die endgültige Veröffentlichung ist für das zweite Quartal 2012 geplant. Für weitere Details und den weiteren Verlauf sei auf die Milestoneplanung in [27] verwiesen.

4.3 Verfügbare Frameworks

4.3.1 RESTeasy

Da die Verwendung des RESTeasy-Frameworks bereits in den Abschnitten 3.2.1.2 und 3.2.2.1 ausführlich an einem konkreten Fallbeispiel beschrieben ist, soll der Vollständigkeit halber an dieser Stelle lediglich eine knappe Zusammenfassung der Arbeit mit RESTeasy erfolgen.

Client

Gemäß der Dokumentation [29] des RESTeasy-Frameworks werden im Client mittels Java Interfaces die Ressourcenklassen abgebildet. Hierbei werden mittels der Annotation aus der JAX-RS-Spezifikation den Methoden des Java Interfaces URI-Templates, die Quellen der Parameter und die zu verwendende HTTP-Methode zugeordnet. Da die URI-Templates Bestandteile des Java-Interfaces sind, lassen sich diese nur zur Entwicklungszeit anpassen. Die Clientimplementierung ist daher an den Dienst physikalisch gekoppelt (vgl. Abs. 2.1.1). Eine Nutzung der Hypermediainformationen aus den Repräsentationen wird nicht unterstützt, obwohl RESTeasy die Links bei der Deserialisierung in die Geschäftsobjekte aufnimmt, falls in der jeweiligen Klasse ein Attribut vom Typ `RESTServiceDiscovery` enthalten ist.

Die clientseitige Unterstützung von Hypermedia durch RESTeasy ist daher als eher gering zu bewerten. Für weitere Implementierungsdetails eines RESTful HTTP Clients sei auf Abschnitt 3.2.2.1 S. 35ff. verwiesen.

Server

Die Implementierung eines Servers mit RESTeasy erfolgt zum einen mittels der Annotationen aus der JAX-RS-Spezifikation und zum anderen mittels Annotationen, die von RESTeasy bereitgestellt werden. Innerhalb einer Ressourcenklasse werden einzelne Methoden an URI-Templates und HTTP-Methoden gebunden. Im Gegensatz zur Client API ist die Verwendung von URI-Templates in der Implementierung des Dienstes nicht problematisch, da dem Dienst in Bezug auf URIs eine eigene Dynamik zusteht. Durch die RESTeasy spezifischen Annotationen ist eine deklarative generische Verknüpfung von Ressourcen möglich. Die `@LinkResource`-Annotation dient dazu die jeweils annotierte Methode als Atom-Link in die Repräsentationen eines zusätzlich bestimmten Java Typs einzufügen, falls ein optional zu definierendes Entscheidungskriterium erfüllt wird. Abbildung 5 verdeutlicht die Bedeutung der einzelnen Parameter der Annotation.

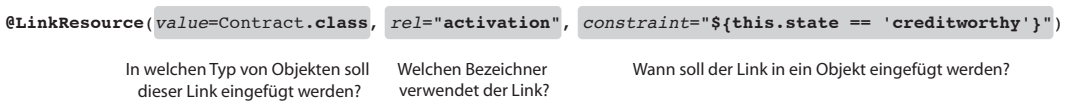

Abbildung 5: Verwendung der @LinkResource-Annotation des RESTeasy-Frameworks

Durch die `@AddLinks`-Annotation wird RESTeasy signalisisiert, dass in die Repräsentation des Rückgabeobjekts einer Methode Links eingefügt werden sollen. Welche Links eingefügt werden, hängt von der Parametrisierung der `@LinkResource`-Annotationen innerhalb aller Ressourcenklassen ab.

Die serverseitige Unterstützung von Hypermedia durch RESTeasy ist positiv zu bewerten, da es dem Anwendungsentwickler durch ein deklaratives Vorgehen ermöglicht wird, das HATEOAS-Prinzip umzusetzen. Die einzige Einschränkung ist, dass Link-Header nach wie vor manuell erstellt werden müssen. In Abschnitt 3.2.1.2 auf Seite 27 sind weitere Details zur Implementierung von RESTful HTTP Diensten vorgestellt.

4.3.2 Spring Web MVC

Das Spring Web MVC-Framework stellt für die Entwicklung sowohl von RESTful Services als auch Clients eine eigene API bereit. Die JAX-RS-Spezifikation wird hingegen nicht verwendet. Stattdessen basiert das Framework auf eigenen Annotationen, die für die Server API jedoch in Namen und Funktion nahezu identisch zu den JAX-RS-Annotationen sind. Begründet ist dies dadurch, dass die Entwickler des Frameworks durch technische Probleme mit der Zusammenführung von JAX-RS und bereits existierendem Spring Web MVC-Code nicht erfolgreich gewesen sind (siehe [38]).

Client

Über ein `RestTemplate` wird die Konsumierung von RESTful Services in der Client API durchgeführt. Dieses Template dient der Kapselung eines HTTP-Clients. Hierzu stellt das RestTemplate die einzelnen HTTP-Methoden bereit, wobei jeder Methode eine URI bzw. ein URI-Template mit den einzusetzenden Parameterwerten übergeben werden muss. Außerdem müssen die Methoden des RestTemplates eine Typinformation erhalten, um zu definieren, welche Art von Objekt in der Antwort erwartet wird. Dies ist notwendig, damit das Unmarshalling der Repräsentation durchgeführt werden kann.

Die Verwendung des `RestTemplate` ist ähnlich zu der des `JdbcTemplate` ausgerichtet (siehe [39]), so dass es lediglich als Abstraktion für den Zugriff auf Daten betrachtet wird. Die Steuerung des Anwendungszustands durch Hypermedia steht nicht im Vordergrund. Daher ergibt sich für die Client API des Spring Web MVC ein ähnliches Problem wie für RESTeasy: URIs werden in der Regel nicht zur Laufzeit entdeckt sondern im Client statisch kodiert. Hierbei handelt es sich um nicht benötigte Abhängigkeiten zur Implementierung des Dienstes.

Um Hypermedia in Kombination mit dem Spring Web MVC-Framework nutzen zu können, muss der Anwendungsentwickler eine eigene Unterstützung implementieren. Das kann bedeuten, dass in den POJO-Klassen zusätzliche Attribute für AtomLinks ergänzt werden, damit die Hypermediainformationen bei der Deserialisierung nicht verworfen werden. Wenn die den Links zugeordneten URIs im Client vorhanden sind, kann das `RestTemplate` statt eines URI-Templates auch eine statische URI verwenden, die ohne Parameter auskommt. Gegenüber RESTeasy ist dies ein Vorteil, da keine neues Template erzeugt werden muss.

Server

Ausgangspunkt für die Implementierung eines RESTful Service sind die Controller -Klassen der Ressourcen. Hierbei handelt es sich um mit der `@Controller`-Annotation versehene Klassen. Innerhalb dieser Klassen werden URI-Templates und HTTP-Methoden durch die `@RequestMapping`-Annotation an die Methoden des Controllers gebunden. In JAX-RS konformen Frameworks werden hierfür mehrere unterschiedliche Annotationen verwendet.

Durch einen im Anwendungskontext definierten sog. „Component Scan“ werden die Ressourcenklassen automatisiert entdeckt und äquivalent zu RESTeasy nach dem *Singleton*-Muster im Application Context instanziiert. Hierbei zeichnet sich ein Unterschied zu Restfulie ab, da dieses Framework standardmäßig einen Request Scope für die Instanziierung benutzt (siehe Seite 52). Aus diesem Grund können die Ressourcenklassen im Spring Web MVC-Framework keinen Zustand halten, sondern müssen den Zustand einer Anfrage durch sämtliche Methodenaufrufe durchreichen. Dies kann zu komplexeren Signaturen der Java-Methoden und damit zu unübersichtlicherem Programmcode führen.

Eine explizite Unterstützung des Hypermedia-Constraints von REST ist in der Server API des Spring Web MVC-Framework nicht vorhanden. Stattdessen müssen Entwickler die Klassen der Geschäftsobjekte so erweitern, dass in diese Links manuell eingefügt werden

können. Innerhalb der Ressourcenklassen muss durch vom Anwendungsentwickler definierte Mechanismen die Sichtbarkeit der Links gesteuert werden.

4.3.3 Restfulie

Restfulie stellt ein Framework zur Entwicklung von Anwendungen auf der Basis von REST-Architekturen dar. Wie bei den anderen Frameworks ist das HTTP die Grundlage für die Kommunikation zwischen Client und Server. Ein wesentlicher Unterschied zu Jersey[12] oder RESTeasy liegt in der Verwendung der JAX-RS-Spezifikation. Dadurch unterscheiden sich sowohl Server als auch Client API von anderen Frameworks. Außerdem soll an dieser Stelle angemerkt werden, dass zu Restfulie keine Dokumentation zur Funktionsweise des Frameworks existiert. Daher basieren die nachfolgenden Erläuterungen auf der Untersuchung des Quellcodes des Frameworks.

Client

Ein zentraler Unterschied in der Client API zu den bisher vorgestellten Frameworks liegt im Verzicht auf eine durch Annotationen gesteuerte Umsetzung der REST-Prinzipien. Die Verwendung eines einzigen Clients für alle Typen von Ressourcen ist bereits in der Client API des Spring Web MVC gezeigt worden. Der `RestClient` wird dazu verwendet eine statische URI wie den Einstiegspunkt aufzurufen. Der RestClient erzeugt ein `Request`-Objekt, welches in mehreren Schritten mit optionalen Parametern konfiguriert werden kann. Um die Anfrage an den Dienst tatsächlich abzuschicken, ruft der Entwickler das Äquivalent der HTTP-Methode an der Request-Instanz auf. Der Apache `HttpClient` wird von Restfulie als low-level Implementierung für die HTTP-Kommunikation verwendet.

Das Ergebnis der Anfrage wird auf der low-level Ebene in Form einer Apache `HttpResponse` dargestellt. Restfulie abstrahiert diese mittels einer eigenen `Response`-Klasse, die durch die `getResource()`-Methode Zugriff auf die deserialisierte Repräsentation der Ressource in Form eines POJOs bietet. Der Entwickler muss beim Methodenaufruf wissen, was für einen Typ er erwartet, da die Methode einen generischen Rückgabedatentyp besitzt.

Um die Hypermediainformationen aus der Dienstantwort nutzen zu können, wird das POJO in der statischen `resource()`-Methode zu einem Objekt vom Typ `Resource` gecastet. Eine `Ressource`-Instanz besitzt beliebig viele Links, die mittels des Relationsbezeichners abgerufen werden können. Von den einzelnen `Link`-Instanzen kann ein `Request`-Objekt erzeugt werden, indem die `follow()`-Methode aufgerufen wird. Die tatsächliche Anfrage wird wie bei dem vom `RestClient` erzeugten `Request`-Objekt durch den Aufruf einer `get()`-, `post()`-, `put()`- oder `delete()`-Methode ausgelöst.

Das folgende Code-Beispiele zeigt, wie sich mit den genannten Klassen und Methoden eine Ressourcenrepräsentation von einem RESTful Service abrufen lässt und wie deren Hypermediainformationen für weitere Interaktionen genutzt werden können.

[12] Jersey ist die Referenzimplementierung zu JAX-RS

```
Response response = restfulie.at("http://localhost:8080/mystore/items")
                                .accept("application/xml").get();
List items = response.getResource();
response = resource(items.get(0)).getLink("self").follow().get();
```

Eine Besonderheit bei der Implementierung eines Clients mit Restfulie stellt der Umgang mit den Hypermediainformationen dar. Die unverarbeitete Antwort vom Webservice wird in Header-Informationen und das übertragene Entity aufgetrennt. Das Entity entspricht der Ressourcenrepräsentation und enthält daher auch Hypermediainformationen. Mittels der XStream Bibliothek werden sowohl XML- als auch JSON-Repräsentationen verarbeitet. Ein von Restfulie eingesetzter LinkConverter dient dabei dem XStream Parser die Hypermediainformationen von den Informationen des POJOs zu trennen. Der LinkConverter ist auf Links im Atom Format ausgerichtet.

Restfulie ist so entworfen, dass aus dem Antwortobjekt ein POJO entnommen werden kann[13]. Damit dieses POJO Hypermediainformationen bereitstellen kann ohne dafür Änderungen am Quellcode der POJO-Klasse durch Anwendungsentwickler zu erzwingen, wird vom Restfulie-Framework mittels eines sog. Enhancers zur Laufzeit ein Subtyp der POJO-Klasse generiert, der zusätzlich das von Restfulie definierte `Resource`-Interface implementiert. Diese generierte Klasse wird mit den POJO- und Link-Informationen instanziiert. Die Instanz wird anschließend zusammen mit den Header-Informationen innerhalb eines `Response`-Objekts zusammengefasst.

Durch den generischen Rückgabedatentyp der `getResource()`-Methode des `Response`-Typs, muss der Anwendungsentwickler den exakten generierten Datentyp des zurückgegebenen Objekts nicht kennen. Dadurch ist es möglich das `Resource`-Interface zu verstecken und erst bei Bedarf durch einen expliziten Cast mittels der statischen `ressource()`-Methode wieder zugreifbar zu machen.

Dieses komplex erscheinende Vorgehen bietet für Entwickler den Vorteil, dass die POJO-Klassen nicht mit Attributen oder Methoden für Hypermediainformationen „verunreinigt werden wie es beim Spring Web MVC- oder RESTeasy-Framework der Fall ist. Ebenfalls wird aus der Perspektive des Entwicklers die Vererbungshierarchie flach gehalten, was zur Übersicht des Programmcodes beiträgt.

In Abbildung 6 auf Seite 53 werden die Abläufe und Zusammenhänge innerhalb eines Hypermedia-bewussten Clients auf der Basis von Restfulie nochmals visualisiert.

Server

Der serverseitige Umgang mit Hypermedieinformationen durch Restfulie unterscheidet sich von den Implementierungen aus RESTeasy oder der Jersey-Erweiterung von Hadley, welche beide JAX-RS als Basis verwenden und um fehlende Hypermediafeatures erweitern.

Ausgangspunkt für die Implementierung einer Ressource ist ein Controller des Vraptor-Frameworks. Ein solcher Controller stellt das Gegenstück zu den Ressourcenklassen aus JAX-RS bzw. den Controllern des Spring Web MVC-Framework dar. Zweck des Controllers

[13] im genannten Beispiel ist dies eine Liste

ist die Abbildungseinheit von HTTP-Anfragen auf Java-Methoden.

Statt der JAX-RS-Annotationen werden von Restfulie die Annotationen des Vraptor-Frameworks verwendet, um mittels URI-Templates und HTTP-Methoden-Annotationen den einzelnen Controllern die eingehenden HTTP-Anfragen zuzuweisen. Das Restfulie-Framework erfordert, dass die Controller-Klassen mit einer `@Resource`-Annotation ausgezeichnet sind, um die Klasse als Einstiegspunkt für Anfragen zu markieren. Für jede eingehende Anfrage werden die jeweiligen Ressourcen-Controller instanziiert und nach der Beantwortung wieder zerstört (*Request Scope*).

Ein Unterschied zum RESTeasy-Framework ist der Verzicht auf einen Rückgabedatentyp bei den Ressourcen-Controller-Methoden. Statt dessen wird bei der Instanziierung des Controllers ein Vraptor `Status`- und ein `Result`-Objekt in den Controller injeziert, die während der Abarbeitung der jeweils aufgerufenen Methode mit Informationen wie etwa der Ressourcenrepräsentation oder Statuscodes ausgefüllt werden.

```
@Resource
public class ThingController {
  @Get
  @Path("/things/{thing.id}")
  public void getThing(Thing thing) {
    thing = database.getThing(thing.getId());
    result.use(representation()).from(thing);
    status.ok();
  }
}
```

Im Gegensatz zu RESTeasy wird die Steuerung der Hypermediainformationen nicht im Controller bzw. in der Ressourcenklasse vorgenommen, sondern erfolgt in den Klassen der Domänen- bzw. Geschäftsobjekte. Aus diesem Grund müssen diese Klassen von Vraptor bereitgestellte `HypermediaResource`-Interface implementieren[14]. Dieses Interface schreibt vor, dass eine `getRelations()`-Methode alle Verknüpfungen für das konkrete Geschäftsobjekt zurückgeben muss. Im Übrigen verwenden die Klassen der Domänenobjekte zur Konfiguration des XML-Bindings statt den JAXB-Annotationen die Annotationen der XStream-Bibliothek.

Am `Restfulie`-Parameterobjekt der `getRelations()`-Methode wird mittels der `relation()`-Methode ein `RelationBuilder` angelegt. Durch den Parameter der `getRelations()`-Methode wird der Relationsbezeichner festgelegt. Der zurückgegebene `RelationBuilder` ermöglicht es, die URI des Links entweder aus einem statischen Parameter zu entnehmen oder durch die Auswertung der Annotationen an den Ressourcen-Controller-Methoden zu generieren. Bei letzterem Ansatz wird die jeweils zu verlinkende Java-Methode an der Controller-Klasse mit den jeweiligen Parametern aufgerufen. So ist z.B. der Link von einer Ressource auf sich selbst eine *GET*-Anfrage an die eigene Controller-Klasse. Das folgende Codebeispiel zeigt diese Umsetzung in reduzierter Form.

```
@XStreamAlias("thing")
```

[14] Restfulie baut auf Vraptor auf

```
public class Thing implements HypermediaResource {
  private String id;
  public List<Relation> getRelations(Restfulie control) {
    control.relation("self").uses(ThingController.class).get(this);
  }
}
```

Falls der Link-Header des HTTP zur Speicherung der Hypermediainformationen verwendet werden soll, müssen diese Informationen manuell in der Controller-Klasse in das `result`-Objekt eingesetzt werden.

Restfulie stellt damit sowohl für Client- als auch Dienstimplementierung eine gute Unterstützung für die Umsetzung des HATEOAS-Prinzips bereit. Einziger Nachteil ist, dass in der Dienstimplementierung die Klassen der Geschäftsobjekte mit Anwendungslogik belegt werden und somit nicht mehr als POJO-Klassen betrachtet werden können.

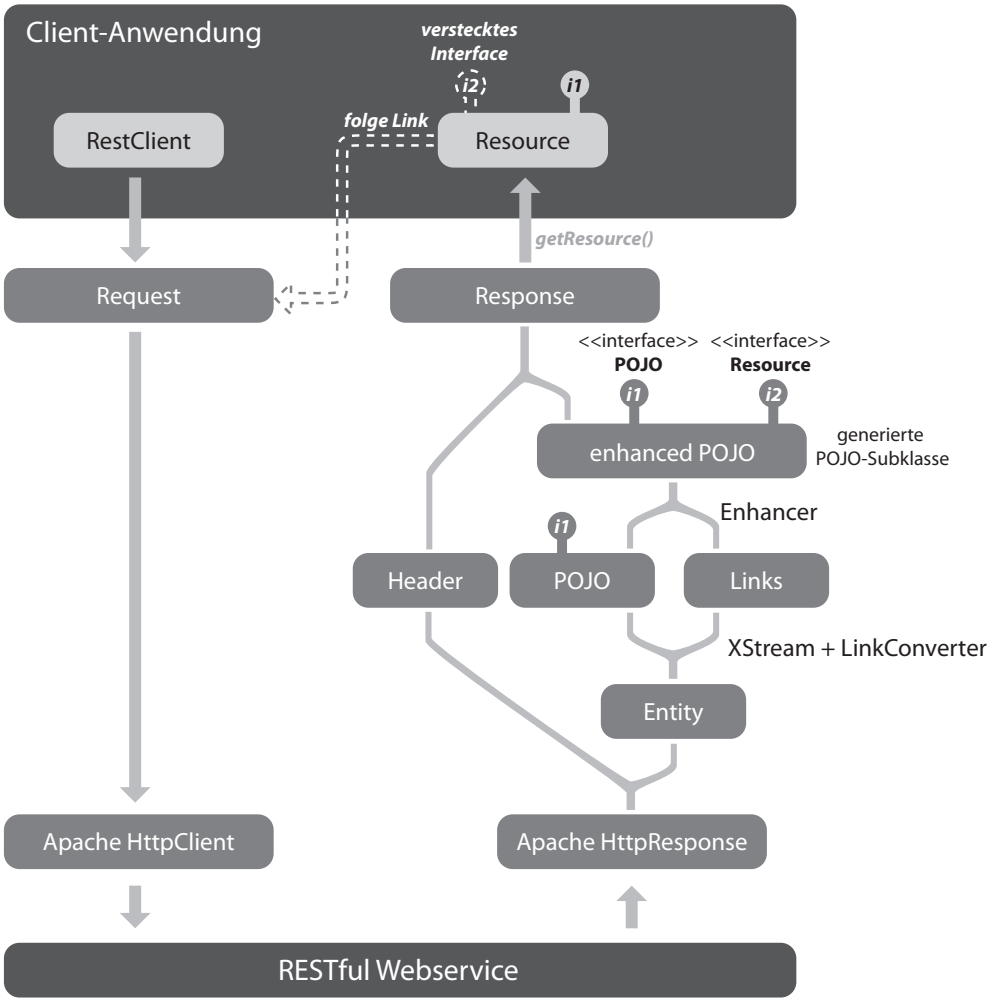

Abbildung 6: Kommunikationsablauf und Umgang mit Hypermedia im Restfulie-Framework

4.3.4 Jersey-Erweiterung von Hadley et al.

Das Jersey-Framework stellt die Referenzimplementierung zur JAX-RS-Spezifikation dar. Wie bereits in Abschnitt 4.2 erwähnt, bietet JAX-RS 1.1 wenig Unterstützung für den Umgang mit Hypermedia. Hadley ist an der Entwicklung von JAX-RS und Jersey beteiligt, so dass er zum Zeitpunkt der Entstehung dieser Arbeit bereits in der Entwicklung der kommenden JAX-RS 2.0 Spezifikation involviert ist.

Im Vorfeld zur Entwicklung von JAX-RS 2.0 stellt Hadley mit seiner Arbeit „Exploring hypermedia support in Jersey“ [16] eine experimentelle Erweiterung des Jersey-Frameworks vor, die eine Unterstützung von Hypermedia durch das Framework mittels Annotations einführt.

Hadley stellt in seiner Arbeit heraus, dass komplexe Workflows, die über CRUD (Create Read Update Delete) einzelner Ressourcen hinausgehen, mittels sog. „action resources“ umgesetzt werden. Der Begriff wird von Hadley wie folgt definiert:

> *„An action resource is a sub-resource defined for the purpose of exposing workflow-related operations on parent resources.“*[16]

Im Fallbeispiel in Abschnitt 3.2.1.2 stellt die Vertragsaktivierung eine solche Aktionsressource dar.

Zwischen dem Dienst und seinen Konsumenten wird ein informeller Vertrag definiert, der die Menge der verfügbaren Aktionsressourcen festlegt. Ein solcher Vertrag ist notwendig, da ein maschineller Konsument wissen muss, nach welchen Aktionen er suchen kann. Hadley formuliert dies wie folgt: *„It follows that at least a subset of the server's state machine—of which actions are transitions—must be statically known for the client [bot] to accomplish some pre-defined task.“* [16]

Die zu einem Zeitpunkt tatsächlich verwendbare Menge von Aktionen ergibt sich aus dem Zustand der Ressource. Um dies umzusetzen, müssen nach REST Hypermediainformationen verwendet werden. Hadley bezeichnet diesen Zusammenhang als kontextuellen, dynamischen Vertrag, da der Anwendungszustand Auswirkungen auf den Vertrag hat. Unter Dynamik versteht er die Tatsache, dass die Details des Vertrags erst zur Laufzeit vom Dienst abgerufen werden. Der statische Teil des Vertrags ist die im Client kodierte Menge der verfügbaren Aktionen bzw. Aktionsressourcen.

Server

Die Server API wird von Hadley um drei Annotationen erweitert: `@Action`, `@ContextualActionSet` und `@HypermediaController`.

Die `@Action`-Annotation markiert Methoden der Ressourcenklassen als Aktionsressourcen bzw. Subressourcen. Als Parameter wird der Annotation der Name der Aktion geben, welcher in einem Link als Relationsbezeichner verwendet wird. Da die JAX-RS-Spezifikation erweitert wird, wird nach wie vor mittels der `@Path`-Annotation das URI-Template der Aktionsressource festgelegt.

Vergleichbar zu Restfulie muss der Anwendungsentwickler eine Methode implementieren, die entscheidet, welche Links abhängig vom Ressourcenzustand in die Ressourcenrepräsentation aufgenommen werden sollen. Im Gegensatz zu Restfulie wird diese Methode nicht in den Klassen der Geschäftsobjekte, sondern innerhalb der Ressourcenklasse in einer mit der `@ContextualActionSet`-Annotation markierten Methode implementiert. Ein weiterer Unterschied zu Restfulie ist, dass die Ressourcenklasse kein spezielles Interface implementieren muss. Dieses Vorgehen bietet den Vorteil, dass die Geschäftsobjekte als POJOs implementiert werden können und keine Logik besitzen.

Damit die soeben erläuterten Annotationen vom Framework ausgewertet werden, muss die jeweilige Ressourcenklasse mit der `@HypermediaController`-Annotation markiert werden. Eine weitere Aufgabe dieser Annotation ist es, festzulegen, welche Art von Links verwendet werden soll. Innerhalb der experimentellen Implementierung von Hadley stehen lediglich Links im Antwort-Header zur Verfügung, da diese für Hadley aufgrund ihrer Formatunabhängigkeit einen geringeren Implementierungsaufwand zur Folge haben. Zu letzt wird in der `@HypermediaController`-Annotation festgelegt, welche Java-Klasse von Objekten durch den Controller exponiert wird.

Client

Die Jersey-Erweiterung von Hadley führt drei Annotationen in die Client API ein. Das Muster der Generierung eines Client-Proxy aus einem Ressourcen-Interface wird von der JAX-RS-Spezifikation übernommen. Analog zur Server API muss das Ressourcen-Interface mit der `@HypermediaController` versehen werden, um es für das Framework erkenntlich zu machen.

Die Methoden des Interfaces werden mittels der `@Action`-Annotation aus der Server API an Stelle der `@Path`-Annotation aus der JAX-RS-Spezifikation annotiert. Damit werden die Methoden des clientseitigen Ressourcen-Interfaces nicht mehr an URI-Templates, sondern an Linkbezeichner gebunden. Ein ähnliches Vorgehen wird in der Vorstellung eines RESTful RESTeasy-Client in Abschnitt 3.2.2.2 auf Seite 37 dieser Arbeit thematisiert.

Da ein Dienst z.B. über Query-Parameter zusätzliche Informationen zur Bearbeitung der Anfrage erhalten kann, markiert Hadley mittels einer `@Name`-Annotation Parameter, die dynamisch an zur Entwicklungszeit nicht bekannten Stellen der Anfrage eingefügt werden. Tilkov weist darauf hin, dass der Begriff „Query-Parameter“ an keiner Stelle in der Dissertation [14] von Fielding auftaucht. Es handelt sich daher um eine für den Architekturstil nicht bewusst vorgesehene Erweiterung.

Um eine tatsächliche Anfrage durchzuführen, fordert der Client beim Dienst ein dem Link zugeordnetes WADL-Fragment an, das den Aufbau der eigentlichen Anfrage beschreibt. Innerhalb dieser Beschreibung ist die zu verwendende URI, die jeweilige HTTP-Methode und die Position der Parameter enthalten. Hadley argumentiert, dass die Verwendung von `@GET/POST/...`- und `@QueryParam`-Annotationen im Interface einen statischen Vertrag wiederspiegelt. Durch seinen „contract-on-demand“-Ansatz wird der Vertrag durch die vom Dienst zur Laufzeit erhaltene Beschreibung dynamischer. Gleichzeitig ist der Client

jedoch darauf angewiesen, dass der Dienst in der Lage ist, diese Informationen bereitzustellen.

Die clientseitige Verwaltung der Links wird durch das Framework übernommen. Jede Interaktion mit dem Dienst führt zu einer Antwort vom Dienst, in der durch Links die zur Verfügung stehenden nächsten Aktionen aufgelistet sind. Auf diese Weise wird während der Kommunikation mit dem Dienst im Client immer das momentan korrekte „contextual action set“ im Client-Proxy gespeichert. Sollten Clientinteraktionen zu einem nicht vorgesehenen Aufruf am Ressourcen-Interface führen, so wirft das Framework eine client-seitige Exception.

Kritik

Hadleys Ansatz kann dahingehend kritisch betrachtet werden, dass zumindest die HTTP-Methoden statisch im Interface kodiert werden können, da diese eine feste Semantik besitzen und nicht frei durch den Dienst verändert werden sollten. Wenn sich die Semantik eines Dienstes ändert, sollte auch der Client sich über diese Bedeutungsänderung bewusst sein und ggf. durch den Entwickler angepasst werden.

Ein weiterer kritischer Aspekt ist, dass die Client API der vorgestellten Jersey-Erweiterung auf die Bereitstellung der WADL-Informationen durch den Dienst angewiesen ist und damit eine engere physikalische Kopplung herstellt. Bereits existierende und ggf. mit anderen Frameworks implementierte Dienste sind zur Client API der Jersey-Erweiterung nicht kompatibel, da eine zusätzliche Abhängigkeit zum Dienst existiert.

Hinsichtlich des Laufzeitverhaltens bewirkt Haldeys Ansatz der dynamischen Verträge rechnerisch eine Verdopplung der Anfragen an den Dienst.

Dadurch, dass Hadley lediglich Verknüpfungen in den Antwort-Headern verwendet, umgeht er die Herausforderung der Umsetzung von in den Repräsentationen eingefügten Verknüpfungen. Ein typischer Anwendungsfall sind z.B. Listenressourcen, die auf mehrere Ressourcen des gleichen Typs verweisen können. In solchen Fällen tauchen Linkbezeichner üblicherweise mehrfach auf, so dass es zu Verwechslungen kommen kann, wenn diese Verknüpfungen extern zur Ressourcenrepräsentation übertragen werden.

4.4 Vergleich der Client-Stile im Umgang mit Hypermedia

Die vorgestellten Frameworks besitzen einige Gemeinsamkeiten aber auch Unterschiede. Nun sollen die unterschiedlichen Stile der Client APIs im Umgang mit Hypermediainformationen miteinander verglichen werden. Die Server APIs werden nicht näher betrachtet.

Sowohl Restfulie als auch die Jersey-Erweiterung von Hadley besitzen dezidierte Mechanismen, um Hypermedia zur Steuerung des Anwendungszustands zu nutzen. Restfulie verbirgt die Relationen eines Objekts zu anderen Ressourcen hinter einem dem Anwendungsentwickler nicht sichtbaren Interface, welches erst durch einen expliziten „Cast“ nutzbar wird. Aus den Relationen lassen sich Links und anschließend Anfragen (Requests) erzeugen. Die Jersey-Erweiterung hält die Detailinformationen der Verknüpfungen nicht im Client, sondern fragt die zu verwendenden URIs, HTTP-Methoden und Parameter zur Lauf-

zeit beim Dienst an. Der Client speichert lediglich den Aktionsbezeichner und optional die Namen der Anfrageparameter.

Sowohl das RESTeasy als auch das Spring Web MVC-Framework bieten für die Entwicklung von Client keine explizite Unterstützung im Umgang Hypermediainformationen. Es ist dem Anwendungsentwickler jedoch möglich, diese Verknüpfungen manuell auszulesen und bei der Erzeugung von Anfragen zu berücksichtigen. Im Spring Web MVC bedeutet dies, dass ein unparametrisiertes URI-Template als Parameter der jeweiligen Methode am RestTemplate verwendet werden muss. Da RESTeasy das URI-Template hingegen im Ressouren-Interface statisch kodiert, muss ein Anwendungsentwickler auf URI-Templates verzichten und für jede Anfrage einen neuen Proxy erzeugen, der die zu verwendende URI als Basis benutzt.

Ein weiteres Unterscheidungskriterium stellt das Wissen des Clients über den Service zur Umsetzung des HATEOAS-Prinzips dar. Dabei geht es um die Frage, mittels welcher Informationen sich ein Konsument vom jeweiligen Dienst abhängig macht. Hier verhalten sich das RESTeasy und Spring Web MVC-Framework gleich. Beide Frameworks erfordern, dass im Client URI-Templates und die zu verwendende HTTP Methode gepspeichert sind, d.h. beide Frameworks ignorieren Hypermediainformationen und machen sich von den URIs der Ressourcen abhängig.

Das Restfulie-Framework unterscheidet sich dadurch davon, dass statt der URI-Templates die Linkbezeichner im Client gespeichert werden. Die zu verwendenden HTTP Methoden sind wie bei RESTeasy und Spring Web MVC statisch im Programmcode festgehalten. Die Jersey-Erweiterung von Hadley verwendet wie Restfulie ebenfalls Linkbezeichner (Aktionsbezeichner) statt URI-Templates, jedoch werden hier im Client keine Informationen über die zu verwendende HTTP-Methode kodiert. Stattdessen wird mittels des Linkbezeichners eine Interaktion bezeichnet, deren konkrete Modalitäten beim Dienst im WADL-Format abgerufen werden (contract-on-demand). Dadurch ist diese Client API auf eine spezielle Dienstimplementierung angewiesen, die diese Informationen bereitstellt.

Die vorgestellten Frameworks folgen unterschiedlichen Paradigmen, wie die Interaktionen mit dem Dienst ablaufen. Die beiden JAX-RS basierten Frameworks RESTeasy und Jersey verwenden Proxy-Objekte, die das Ressourcen-Interface implementieren. Die Methodenaufrufe am Proxy-Objekt werden durch frameworkspezifische Mechanismen auf HTTP-Anfragen abgebildet. Das Spring Web MVC-Framework basiert auf dem Paradigma eines universellen Templates, um Objekte abzurufen, zu manipulieren oder zu zerstören. Restfulie verwendet ein ähnliches Paradigma, jedoch wird das universelle Template (im Fall von Restfulie das Request-Objekt) aus den Relationen der Geschäftsobjekte für jede Interaktion neu erzeugt, anstatt ein einzelnes für sämtliche Interaktionen zu verwenden.

In Abbildung 7 werden die Stile der Client APIs visualisiert, um deren - ggf. fehlenden - Umgang mit Hypermediainformationen zu zeigen.

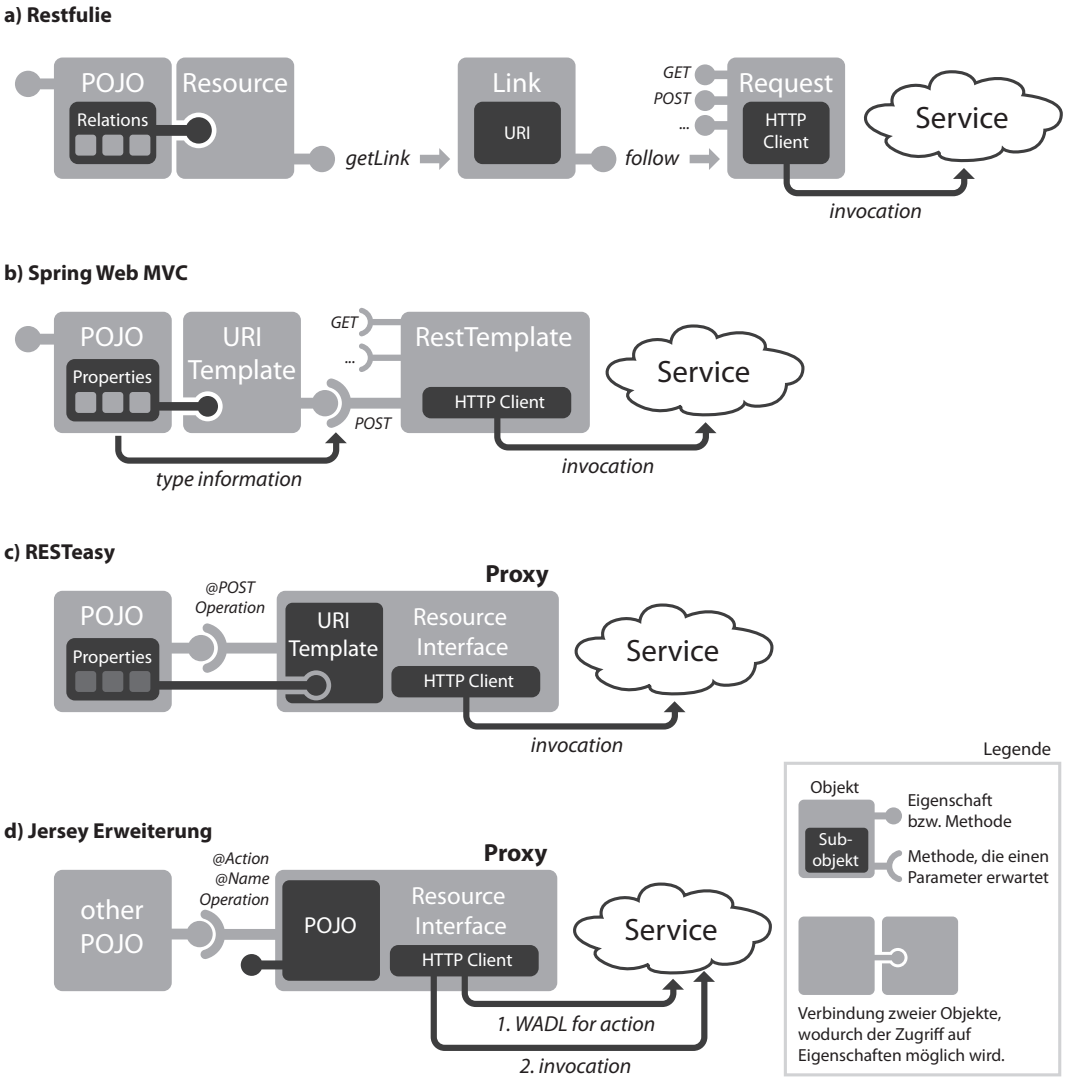

Abbildung 7: Stile der Clients in unterschiedlichen Frameworks: a) Erweiterung der POJOs um Hypermediainformationen b) universeller Client mit URI-Templates c) operationsorientierter Proxy mit URI-Templates d) operationsorientierter Proxy mit „contract-on-demand“

5 Komposition von RESTful Services

Im vorangegangen Abschnitt ist gezeigt worden, wie unterschiedliche Frameworks die Bereitstellung bzw. Nutzung von Hypermediainformationen unterstützen. Neben individuell entwickelten Clients können auch die Laufzeitumgebungen von Dienstkompositionen als Clients für RESTful Services betrachtet werden.

Der Themenbereich der Dienstkomposition wird durch viele Arbeiten untersucht, die sich dabei jedoch auf die Orchestrierung von Diensten auf der Basis der WS*-Technologien fokussieren. Üblicherweise wird zur Orchestrierung dieser Services die BPEL eingesetzt. BPEL selbst ist an die Verwendung von WSDL 1.1 und SOAP gebunden, da BPEL für diesen speziellen Zweck entwickelt wurde (siehe Abs. 2.4.2). Durch die Verwendung von visuellen Editoren sollen Personengruppen wie z.B. die Unternehmensfachseite in die Lage gebracht werden, die in der Schnittstellenbeschreibung spezifizierten Operationen und Endpunkte zu ausführbaren Prozessmodellen zusammenzusetzen. Ein tieferes technisches Verständnis des Prozessdesigners für die Implementierung der Operationen wird dabei auf ein Minimum reduziert, da die Dienste im Idealfall nach den Vorgaben des Unternehmensfachseite entwickelt und dokumentiert sind.

Die Komposition von RESTful Services rückt erst seit wenigen Jahren in den Vordergrund der Forschung. Dabei müssen sich Entwickler mit Problemen auseinandersetzen, die bei der Verwendung von SOAP Services nicht vorhanden sind. So ist bei der Verwendung von RESTful Services keine maschinenlesbare Schnittstellenbeschreibung vorgesehen, um die Kopplung der Dienstkonsumenten an den Dienst zu reduzieren. Stattdessen sollen sich Clients durch die Kontrollflussinformationen innerhalb der Ressourcenrepräsentationen steuern lassen, wie es in der Nutzung des „human web“ der Fall ist.

Der Umgang mit Kontrollflussinformationen erfordert ein korrektes Verständnis des HATEOAS-Prinzips von REST. Zwar wird in Arbeiten von Alarcon et al. [4] und Pautasso [34] die Aufgabe von Hypermedia korrekt wiedergegeben, jedoch kommt es in den Fallbeispielen ihrer Ansätze durch Verletzung des HATEOAS-Prinzips zu einer stärkeren Kopplung zwischen Dienst und Client als nötig. Ursache hierfür ist, dass zu viel Wissen über den Dienst in den Clients kodiert wird. Die häufigste Ursache für überhöhte Kopplung zwischen Client und Service innerhalb einer nach REST-Architektur erfolgt durch die Kodierung des Aufbaus von URIs in den Clients. Diese Problematik ist in Abschnitt 4.4 bereits näher diskutiert worden.

Zwar liegt es nahe, dass Client-Entwickler URI-Templates für Ressourcen verwenden, da diese eine generische Art und Weise bieten, um die Ressourcen zu adressieren. Allerdings kann sich ein Client durch die Loslösung von den Kontrollflussinformationen des Dienstes entgegen der Geschäftsregeln verhalten und zu jedem Zeitpunkt mit jeder beliebigen Ressource arbeiten. Daher muss bei der Komposition von RESTful Services ein Fokus auf der Nutzung Hypermediainformationen gesetzt werden.

Die Orchestrierung von RESTful Services stellt Anwendungsentwickler vor das Pro-

blem, dass zwei Instanzen über den Anwendungsfluss entscheiden wollen. Zum einen legt der Dienst fest, welche Aktionen auf einer Ressource durchgeführt werden können und zum anderen will die steuernde Einheit der Komposition einen bestimmten Zustand der Ressourcen erreichen, indem ein zuvor definierter Anwendungsfluss eingehalten wird. Hierbei handelt es sich jedoch nicht um einen scheinbaren Konflikt. Der Prozessentwickler muss sich bei der Definition seines Anwendungsflusses lediglich an die Regeln des Dienstes halten. D.h. er muss mit den Hypermediainformationen arbeiten, die ihm der Dienst anbietet. Wenn Änderungen des Dienstes dazu führen, dass das Prozessmodell nicht mehr gültig ist, so darf sich der Prozessentwickler nicht über diese Änderungen hinwegsetzen und auf statisch kodierte URIs zurückgreifen.

Bei der Choreografie von RESTful Services müssen sich die Anwendungsentwickler mit der gleichen Problematik beschäftigen. Hierbei ist jedoch der Unterschied, dass die Rolle des Prozessentwicklers auf die Entwickler der einzelnen Dienste übertragen wird.

Tilkov hebt in [45] hervor, dass es zur Zeit keinen akzeptierten Standard für die Orchestrierung von RESTful HTTP Services gibt. Darüber hinaus ist seine persönliche Auffassung, dass die Bedeutung der Orchestrierung oftmals überbewertet sei. Ist die Möglichkeit zur Orchestrierung von Diensten jedoch eine zwingend erforderte Anforderung, so muss aus den bisherigen Lösungen ausgewählt werden. Daher sollen im Folgenden unterschiedliche Arbeiten zu diesem Thema in Abschnitt 5.1 näher betrachtet und ihre Vorzüge bzw. Schwachstellen in Abschnitt 5.2 herausgestellt werden.

5.1 Ansätze

Der älteste der betrachteten Ansätze stammt von Pautasso. Er stellt in seiner Arbeit „BPEL for REST“ [33] eine *Erweiterung von BPEL* vor, die es ermöglicht in der Prozessbeschreibung auf Ressourcen via HTTP zuzugreifen. Vor dem Hintergrund der starken Verbreitung von BPEL ist dieser Ansatz nachvollziehbar. Deshalb soll er im nächsten Abschnitt näher behandelt werden.

Eine weitere Arbeit von Pautasso widmet sich einem generischeren Ansatz, der eine weniger auf bestimmte Architekturen ausgerichtete Komposition von Diensten zu unterstützen versucht. Hierzu stellt Pautasso an Hand von *JOpera* ein generischeres Werkzeug zur Orchestrierung von Diensten vor.

Das Fehlen einer maschinenlesbaren Beschreibung von RESTful Services ist Thema einer Arbeit von Alarcon et al.. Die Beschreibung von Ressourcen mittels WADL greift Aspekte der WSDL auf, wobei die von den Autoren vorgestellte *Resource Linking Language* (ReLL) auf den Hypermedia-Constraint von REST ausgerichtet ist und diesen nicht zu verletzen versucht. Da die Orchestrierung ein Einsatzgebiet der ReLL ist, wird von den Autoren die Funktionsweise mittels eines als Petri-Netzes modellierten Prozesses demonstriert.

5.1.1 Erweiterung von BPEL

Die Erweiterung von BPEL um Sprachkonstrukte für die Unterstützung von RESTful HTTP Services ist die Kernidee von Pautassos Arbeit „BPEL for REST“.

Als zentrales Problem der Kombination von BPEL und REST stellt Pautasso die Diskrepanz zwischen dem Verlangen von statischen Dienstbeschreibungen auf der Seite von BPEL und der Dynamik der URIs auf der Seite von REST heraus. Ziele der Erweiterung von BPEL sind die *Konsumierung von RESTful HTTP Services*, welche eine Kompensation der fehlenden maschinenlesbaren Dienstbeschreibung erfordern, und die Erzeugung neuer *RESTful Services als Ergebnis der Komposition*.

5.1.1.1 Vorüberlegungen

Die Idee mittels Verwendung von WSDL 2.0 die fehlende Beschreibung von RESTful Services nachzureichen, wird von Pautasso aus zwei Gründen verworfen:

1. WSDL 2.0 ist in der Praxis nur wenig verbreitet und wird durch BPEL nicht unterstützt. Eine Erweiterung von BPEL auf den WSDL 2.0 Standard ist laut Pautasso nicht absehbar. In der Praxis existieren jedoch BPEL-Engines[15], die sowohl nach dem WS-BPEL 2.0 Standard und WSDL 1.1 als auch zusätzlich WSDL 2.0 arbeiten können. Diese Unterstützung von WSDL 2.0 ist jedoch nicht Teil des offiziellen OASIS Standards (siehe [30]).

2. Die fehlende maschinenlesbare Dienstbeschreibung muss vom BPEL-Entwickler nachträglich erstellt werden. Dies führt zu einer Verlagerung des Entwicklungsaufwands vom Dienstanbieter hin zum Dienstkonsumenten (siehe [33]).

Die Möglichkeit, eine alternative Beschreibungssprache wie etwa WADL zu verwenden, wird von Pautasso zwar angesprochen, jedoch sind nur in seltenen Fällen für RESTful Services WADL Beschreibungen verfügbar (siehe [33]). Außerdem ist WADL nicht darauf ausgelegt, das HATEOAS-Prinzip zu unterstützen (siehe Abs. 2.3.2). Daher verzichtet Pautasso auf eine vom Prozessmodell entkoppelte Dienstspezifikation und macht die Dienstbeschreibung zu einem impliziten Bestandteil des BPEL Prozessmodells. Hierfür wird BPEL um Sprachkonstrukte für den Aufruf der RESTful HTTP Services erweitert. Dies bedeutet, dass die fehlende Unterstützung von WSDL 2.0 in BPEL oder das Fehlen von Dienstbeschreibungen kein unkompensierbares Problem darstellt.

Die Verschiebung des Aufwands der Erstellung einer wenn auch impliziten Dienstbeschreibung ist jedoch nach wie vor vom Anbieter hin zum Konsumenten vorhanden. Das bedeutet, dass sich die Entwickler des Prozessmodells (z.B. die Unternehmensfachseite) mit den Implementierungsdetails des Dienstes und Konzepten von REST auseinandersetzen muss.

Der Verzicht auf eine formale Dienstbeschreibung hat trotz der Aufwandsverlagerung den Vorteil, dass den Hypermediainformationen mehr Beachtung gewidmet werden muss.

[15] z.B. EasyBPEL http://easybpel.petalslink.org/index.html

Statt auf URI-Templates innerhalb eines WADL- oder WSDL 2.0 Dokuments zurückzugreifen, kann in einer erweiterten Version von BPEL eine Zustandssteuerung durch Links innerhalb der Ressourcenrepräsentationen erfolgen. Pautasso leitet daraus für die BPEL-Erweiterung die Anforderung ab, dass ein Mechanismus zur Extraktion der URIs aus den Repräsentationen notwendig ist, um eine dynamische Bindung von den Aktivitäten des Prozessmodells zu den URIs der Ressourcen zu erhalten. Sein Ansatz strebt nach einer *„nativen Unterstützung zur expliziten Kontrolle“* (siehe [33]) der Ressourcen.

5.1.1.2 Umsetzung

In den folgenden Abschnitten soll die Umsetzung der BPEL-Erweiterung vorgestellt und auf Problemstellen hingewiesen werden. Dazu wird stellenweise auf den Beispielprozess aus Pautassos Arbeit [33] hingewiesen. Dieser befindet sich aus Platzgründen im Anhang A.4 und ist aus den einzelnen Codefragmenten aus der genannten Quelle zusammengesetzt.

Request Handler

Da die Komposition von Diensten auf Grund der rekursiven Natur (vgl. Abs. 2.2.2 S. 16) selbst wieder Teil einer weiteren Komposition sein kann, stellt ein Geschäftsprozess in Pautassos Ansatz eine Ressource dar. Während ein Prozess aktiv ist, verändert sich sein Zustand bzw. der Zustand seiner Subressourcen. Damit eine Prozessressource angelegt und der Zustand durch den Prozessnutzer gesteuert werden kann, müssen Interaktionsschnittstellen mittels des uniformen HTTP Interfaces definiert werden. Hierzu führt Pautassos Erweiterung für jede der HTTP-Methoden einen *Request Handler* ein. Ein solcher Handler wird aktiv, sobald die entsprechende HTTP-Methode an der URI der Prozessressource aufgerufen wird. Innerhalb des Handlers wird sowohl mittels der standardisierten BPEL als auch neuer RESTful HTTP-spezifischer Sprachelemente der auszuführende Prozessablauf definiert.

So ist es möglich, innerhalb dieses Flusses neue Subressourcen anzulegen, die wiederum eigene Request Handler besitzen. Ein mögliches Einsatzszenario hierfür ist die Einbeziehung von Menschen in den Prozess. Durch sie durchgeführte Aktivitäten werden dadurch abgeschlossen, dass die jeweilige Person den Zustand der exponierten Subressourcen verändert. So wird in Pautassos Arbeit [33] mittels einer Subressource dem Prozessnutzer die Möglichkeit gewährt, sich während der Prozessausführung zwischen zwei möglichen an einem Prozess beteiligten Dienstanbietern zu entscheiden.

Ein Request Handler ist dafür verantwortlich, die Antwort auf eine Anfrage an die Prozessressource oder einer ihrer Subressourcen zu bestimmen. Hierfür wird von Pautasso die Response-Aktivität definiert, welche u.a. ein Attribut für den HTTP-Statuscode der Antwort enthält. Innerhalb der Response-Aktivität kann der Nachrichteninhalt statisch formuliert oder dynamisch erzeugt werden. Eine Response-Aktivität muss nicht zwangsläufig die letzte Aktivität innerhalb eines Request Handlers sein. Aktivitäten, die auf das Versenden der Antwort folgen, werden von der Prozessengine ebenfalls ausgeführt.

Die Request Handler besitzen unterschiedliche Eigenschaften, die sich aus den jeweiligen HTTP-Methoden ergeben, für die sie bestimmt sind. Da es sich bei *GET* um eine sichere

Methode handelt, führt ein *GET*-Aufruf zu keiner Veränderung am Zustand der Ressource. Um dies sicherzustellen, besitzt ein *onGet*-Request Handler lediglich lesenden Zugriff auf die Variablen der Ressource. *POST* hingegen ist als nicht idempotent definiert, so dass hier eine isolierte Ausführung aktiviert werden kann, um Problemen durch die parallele Verarbeitung von modifizierenden Anfragen entgegenzuwirken. Abbildung 8 veranschaulicht diese Zusammenhänge.

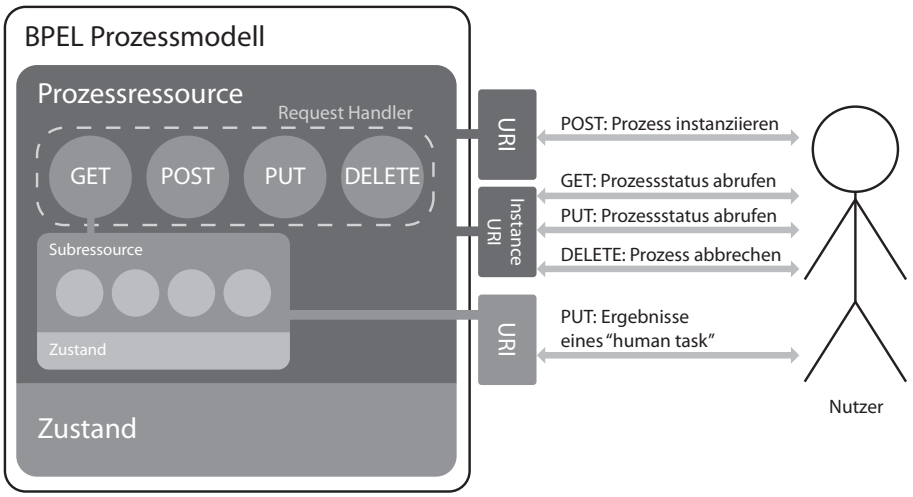

Abbildung 8: Aufbau von „BPEL for REST“-Prozessmodellen

HTTP Aktivitäten

Um RESTful HTTP Services aufzurufen, führt Pautasso in seiner BPEL Erweiterung vier Aktivitäten ein, die auf die HTTP-Methoden *GET*, *POST*, *PUT* und *DELTE* verweisen. Jede der Aktivitäten besitzt ein URI-Attribut, welchem statische Adressen wie z.B. einem Einstiegspunkt oder dynamische Adressen wie z.B. aus Links oder aus berechneten URI-Templates zugewiesen werden können. Die URI-Templates stellen in diesem Fall jedoch lediglich eine Notlösung für nicht RESTful gestaltete Dienste dar.

Jede der Aktivitäten kann durch beliebig viele Header-XML-Elemente erweitert werden, um den HTTP-Header der Anfragen an den Dienst zu konfigurieren. Ein mögliches Anwendungsszenario hierfür ist z.B. die Content-Type-Negotiation mittels des *Accept*-Headers.

Hypermedia

Pautasso fügt in „BPEL for REST“ die in Abschnitt 4.1.1 für einzelne Statuscodes vorgeschriebenen Hypermediainformationen in die Antworten der Prozesslaufzeitumgebung ein. Wird ein Prozess instanziiert, so erhält der Konsument den Statuscode 201 CREATED und setzt den Wert des *Location*-Header auf die URI der erstellten Prozessinstanz (siehe [33] S. 289).

Es existieren jedoch mehrere Aspekte, auf die Pautasso nicht eingeht:

1. Zum einen ist nicht näher beschrieben, wie Subressourcen in der Repräsentation der Prozessressource durch Links für den Dienstkonsumenten sichtbar werden. Pautas-

so schreibt hierzu: „If a [sub]resource is declared from within a local scope of the process, its URI is published only once the execution of a particular process instance reaches the particular scope.“ [33] Der Mechanismus, wie diese URI veröffentlicht wird, ist nicht beschrieben. Es ist daher unklar, ob ein Dienstkonsument wiederholt den Zustand der Prozessressource anfragen muss, um das Vorhandensein eines Links auf die Subressource abzuwarten oder wie der Konsument sonst von der Verknüpfung erfährt. Darüber hinaus wäre es möglich, die URIs der Subressourcen direkt nach der Instanziierung des Prozesses zu veröffentlichen. Wie in Abschnitt 2.5 erwähnt, muss eine URI nicht zwangsläufig auf eine bereits existierende Ressource verweisen.

2. Zum anderen ist nicht klar, auf welche Weise Links aus den Repräsentationen der orchestrierten Dienste extrahiert werden. Ob dabei standardisierte Formate wie etwa Atom zum Einsatz kommen oder ob die Links in individuellen Attributen der Repräsentationen eingebettet sind, ist nicht klar. In seinem Beispielcode verwendet Pautasso eigene Variablen, um die zur Laufzeit entdeckten URIs zu speichern. Die Zuweisung der Werte für diese Variablen erfolgt in Pautassos Beispiel direkt. D.h. ein Parsing der erhaltenen Antworten ist nicht erkennbar. Daher ist davon auszugehen, dass die Hypermediainformationen nicht in generischen Links, sondern als eigene Attribute in den Repräsentationen enthalten sind. Zwar erleichtert dieses Vorgehen die Extraktion von URIs, da keine Suche über eine Mehrzahl von Links notwendig ist, jedoch bietet ein standardisiertes Format eine breitere Unterstützung für eine breitere Menge an Clients. Davon abgesehen konnte in Abschnitt 4.3 gezeigt werden, dass Dienste Hypermediainformation häufig mittels AtomLinks umsetzen. Darüber hinaus können Links auch im HTTP-Header kodiert auftauchen. Auf HTTP-Header geht Pautasso jedoch nur zur Beschreibung des Datenformats einer Dienstantwort ein.

Neben diesen Unklarheiten zeigt sich in Pautassos Beispielprozess aus [33] eine weitere Schwachstelle im Umgang mit Hypermedia. Er verwendet URI-Templates, um die URIs der Ressourcen aus den zu orchestrierenden Diensten aufzubauen, statt über einen zentralen Einstiegspunkt zu den benötigten Ressourcen zu gelangen. Hierdurch entsteht eine starke physikalische Kopplung zwischen den konsumierten Diensten und dem Prozessmodell. Durch einen Mehraufwand ist es vorstellbar, dass durch aufeinanderfolgende Interaktionsschritte die URIs der benötigten Ressourcen zur Laufzeit dynamisch entdeckt werden können.

Eine mögliche Ursache für die Verwendung von URI-Templates ist, dass in der URI Query-Parameter enthalten sind (siehe Abs. 4.3.4 auf S. 55). Die Werte der Parameter müssen durch den Client bestimmt werden, so dass der Client die zu verwendende URI selbst konstruieren muss. Ein direkter Link ist nicht umsetzbar, da der Dienst die Werte der Parameter nicht im Vorfeld kennen kann.

Abschließend kann somit festgehalten werden, dass Pautasso die Dienst-Seite HATEOAS-Prinzip im Bereich der Instanziierung von Prozessen umsetzt. Allerdings verhält sich das Prozessmodell seines Beispiels nicht RESTful, so dass die Client-Seite des

HATEOAS-Prinzips in seinem Ansatz weniger intensiv thematisiert wird.

5.1.1.3 Prozesslebenszyklus

In „BPEL for REST“ wird ein Prozess durch eine *POST*-Anfrage an die URI der Prozessressource instanziiert. Damit mehrere Prozessinstanzen parallel genutzt werden können, müssen sie voneinander unterschieden werden. Laut Pautasso werden hierfür in einem einfachen Fall Sitzungen erstellt, wobei die Sitzungskennung beim Client als Cookie ablegt wird und bei jeder folgenden Anfrage mitgeschicken werden muss.

Ein anderer Ansatz ist es bei der Instanziierung des Prozesses im Location-Header der Antwort eine eindeutige URI der neuen Prozessinstanz zu platzieren. Hierzu muss jedem Prozess eine eindeutige Kennung zugewiesen werden indem z.B. eine Zählervariable verwendet wird. Durch dieses Vorgehen ist es möglich, dass der Client nicht auf eine Sitzung beim komponierten Dienst angewiesen ist.

Betrachtet man beide Möglichkeiten so die letzte Variante die einzig REST-konforme Lösung, da die Anfragen an den komponierten Dienst statuslos übertragen werden. Tilkov schlägt ebenfalls die Umwandlung von Service-Status in Ressourcenstatus als Ausweg aus der statusbehafteten Kommunikation vor (siehe [45] Seite. 108).

Eine Prozessinstanz wird zerstört, entweder wenn keine Aktivitäten mehr ausgeführt werden können oder wenn die *exit*-Aktivität im Prozessmodell erreicht wird. Dabei ist zu beachten, dass die Zerstörung einer Prozessinstanz nicht die Zerstörung des uninstanziierten Prozesses zur Folge hat. Diese existiert nach wie vor, um neue Instanzen erstellen zu können oder bereits existierende Prozessinstanzen weiterarbeiten zu lassen.

Eine Besonderheit der BPEL-Erweiterung von Pautasso ist, dass Ressourcen innerhalb bisher auf WS*-Technologien begrenzte BPEL-Prozesse erzeugt werden können. Mögliche Einsatzszenarien sind die Veränderung des Prozesszustands über eine alternative API oder die Ergebnisse von menschengesteuerten Aktivitäten über eine HTTP-Schnittstelle in den Prozess einzubeziehen.

5.1.2 Ressourcenbeschreibung mittels Metamodell

Alarcon et al. widmen sich in ihrer Arbeit „Hypermedia-Driven RESTful Service Composition“ [4] der Komposition von RESTful Services und konzentrieren sich dabei besonders auf den Hypermedia-Constraint von REST. Grund hierfür ist, dass dieser laut den Autoren nicht ausreichend beachtet wird. Dabei stellen Alarcon et al. heraus, dass Hypermedia nicht nur dazu eingesetzt werden sollte, um Beziehungen zwischen Entitäten aufzuzeigen, sondern auch, um mögliche Zustandsübergänge der Ressourcen freizugeben bzw. zu blockieren (vgl. HATEOAS-Prinzip Abs. 2.5).

Den Autoren gemäß sieht der REST-Architekturstil Menschen als Hauptkonsumenten von Diensten, während Maschine-zu-Maschine-Kommunikation auf Grund einer fehlenden bzw. informellen Dienstbeschreibung erschwert ist. Daher stellen Alarcon et al. die Beschreibungssprache *ReLL* vor, die den Hypermedia-Constraint berücksichtigen soll und somit das Entdecken von Ressourcen zur Laufzeit ermöglicht (*dynamic late binding*). Die

Autoren weisen jedoch auch darauf hin, dass eine Dienstbeschreibung den Grad der Kopplung zwischen Dienst und Client erhöht. Damit trägt auch ReLL zu einer engeren Kopplung bei. Alarcon et al. grenzen ihre Beschreibungssprache dahingehend von WADL ab, dass WADL auf URI-Templates basiert und keine Unterstützung für das HATEOAS-Prinzip bietet. Daher ist WADL eher den operationszentrischen Beschreibungssprachen wie etwa der WSDL zuzuordnen (siehe [4]).

5.1.2.1 ReLL Metamodell

In diesem Abschnitt soll das Metamodell zur ReLL vorgestellt werden, um die Modellierung von Ressourcen und ihren Verknüpfungen mit anderen Beschreibungssprachen vergleichen und bewerten zu können.

In ReLL exponiert ein `Service` eine oder mehrere Ressourcen. Eine `Resource` besitzt eine eindeutige Kennung, einen Namen, eine Beschreibung und ein optionales URI-Template. Der Zustand einer Ressource kann durch mehrere unterschiedliche Repräsentationen beschrieben werden, wobei jede Repräsentation (`Representation`) eine beliebige Anzahl von Links enthalten kann. Ein `Link` verweist auf andere Ressourcen statt auf eine URI (`target`-Assoziation). Auf diese Weise muss der Typ einer Ressource, die durch eine URI referenziert wird, nicht aus der URI erschlossen werden (siehe [3]). Ein `Link` wird mittels medientypspezifischer Selektoren (`Selector`) aus den Repräsentation extrahiert. Für XML-Repräsentationen lassen sich z.B. mittels XPath die URIs der Links aus den Repräsentationen extrahieren. Ein Selektor ist jedoch nicht ausschließlich an den Entity-Bereich einer HTTP-Antwort gebunden. Selektoren können ebenfalls Links aus den Metadaten der Antwort entnehmen (z.B. der Link-Header des HTTP 1.0). Die Umsetzung der Selektoren wird durch Alarcon et al. nicht näher beschrieben. Einem Link kann optional ein Protokoll zugewiesen werden, welches die Methode festlegt, mit der die verknüpfte URI aufgerufen werden muss. Abbildung 9 zeigt das Metamodell als Klassendiagramm aus [4].

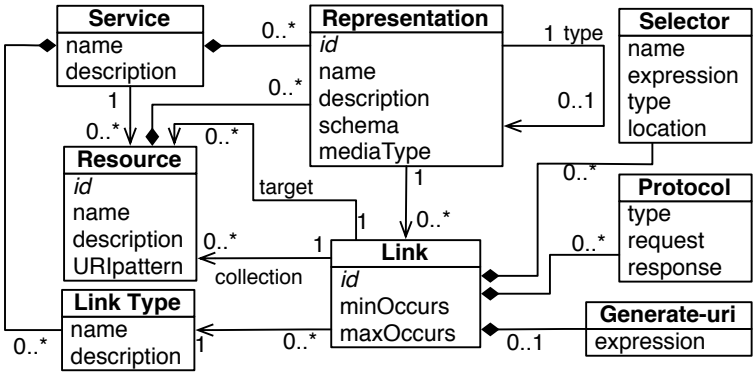

Abbildung 9: ReLL Metamodell aus [4]

Die Besonderheit des ReLL Metamodells stellt die Typisierung der Links dar. Hierdurch werden Bedeutungszusammenhänge der Anwendungsdomäne explizit innerhalb der

Dienstbeschreibung kodiert. Außerdem erlaubt dieses Vorgehen Ressourcenrepräsentationen automatisiert abzurufen und mittels einer Transformation in formal semantische Formate umzuwandeln. Hierfür verweisen Alarcon et al. auf XSLT zur Transformation und das RDF (Resource Description Framework) als semantisches Datenformat (siehe Abschnitt 3 in [4]). Wird ein ReLL-Modell auf nicht semantisch ausgezeichnete Ressourcen angewandt, so lassen sich mit einem Zusatzaufwand SPARQL[16]-Anfragen auf die Ressourcen anwenden.

5.1.2.2 Prozesskoordinaten mittels Petri-Netz

Die Idee, Petri-Netze zur Modellierung von Workflows einzusetzen, existiert bereits seit längerem. So stellen Aalst et al. Petri-Netze als ein mathematisches graphisches Konzept vor, das bereits 1962 als *„Werkzeug zur Modellierung und Analyse von Prozessen“* (siehe [1]) entwickelt wurde. In Bezug auf die Komposition von RESTful Services haben Decker et al. bereits einen Ansatz in [9] auf der Basis von Petri-Netzen vorgestellt. Alarcon et al. zeigen mittels eines Fallbeispiels, wie sich durch ReLL beschriebene RESTful Services durch ein Petri-Netz orchestrieren lassen. Das Petri-Netz ist dabei so entworfen, dass es die Ablauflogik des Prozesses widerspiegelt.

An dieser Stelle soll keine ausführliche Einführung in Petri-Netze erfolgen. Stattdessen werden die unterschiedlichen Teilkonzepte von REST mit Konzepten der Petri-Netze in Beziehung gesetzt. Hierzu führen Decker et al. an, dass der Anwendungszustand eines RESTful Clients durch die Position der Marken innerhalb eines Petri-Netzes repräsentiert wird. Die Transitionen zwischen den Zuständen des Petri-Netzes werden auf Hypermediainformationen abgebildet. Das Absenden einer Anfrage entspricht dem Aktivieren einer Transition. An dieser Stelle unterscheiden sich jedoch die Ansätze von Decker [9] und Alarcon [3], da Decker die Transition einer URI zuordnet, während Alarcon diese den Links zuordnet, die im ReLL Metamodell neben der URI auch noch Informationen über den Typ der referenzierten Ressource und die zu verwendende Methode am Interface besitzen. Die Zustände innerhalb des Petri-Netzes können als die Antworten der RESTful Services verstanden werden[17].

5.1.2.3 Zusätzliche Kopplung

Alarcon et al. verweisen in ihrem Fazit auf eine geringe, aber nach wie vor vorhandene Kopplung zwischen Client und Dienst durch die zusätzliche Ressourcenbeschreibung mittels ReLL. Diese Kopplung ist jedoch auch in anderen Ansätzen wie etwa „BPEL for REST“ vorhanden, da die für Modellierung eines Prozess zu suchenden Verknüpfungen im Client kodiert werden müssen. Damit ist der Client davon abhängig, dass der Dienst die Linkbezeichner nicht verändert. Im Client wird so betrachtet ein Navigationspfad durch das Ressourcennetz beschrieben. Abbildung 10 zeigt an Hand des Beispiels aus Kapitel 3.1, wie ein Client durch Links gesteuert mit den HTTP-Methoden durch die Repräsentationen der an einem Prozess beteiligten Ressourcen navigiert.

[16] SPARQL ist eine Anfragesprache für RDF (siehe [2])
[17] Die Antwort enthält u.U. eine Repräsentation der Ressource.

Die Definition des Pfades muss dem Prozessentwickler ermöglicht werden, indem die Relationen zwischen den Ressourcen verständlich sind. Ob der Prozessablauf auf eine zwischengeschobene Abstraktionsschicht zur Beschreibung der Relationen zwischen den Ressourcen wie etwa ReLL aufgebaut ist oder ob die benötigten Informationen direkt im Prozessmodell vorliegen, ist eine Designentscheidung, die nicht allgemein getroffen werden kann.

Ein möglicher Grund für eine zusätzliche Abstraktionsschicht ist dadurch gegeben, dass technische Details vor dem Entwickler verborgen werden können. Im Fall von ReLL kann ein Link dafür sorgen, dass der Prozessentwickler von der Definition der Methoden für die Dienstaufrufe befreit wird, da diese bereits Teil des ReLL-Links sind. In „BPEL for REST“ hingegen muss der Entwickler selber bewusst entscheiden, welche Methoden auf die URIs der einzelnen Links angewandt werden müssen.

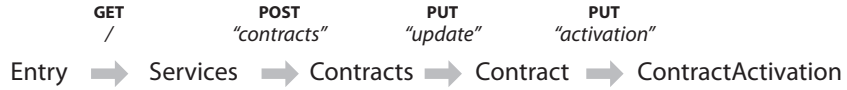

Abbildung 10: Navigationspfad durch den in Kapitel 3.1 beschriebenen Prozess

5.1.2.4 Beziehung zur WSDL

Simon et al. stellen in [42] heraus, dass ein WSDL Dokument nur eine syntaktische Dienstbeschreibung darstellt. Eine Beschreibung der an den Operationen beteiligten Konzepte und den Konsequenzen der Operationen fehlt hingegen.

Die von Alarcon et al. vorgestellte ReLL ist auf die Konsumierung von RESTful Services unter Berücksichtigung des Hypermedia-Constraints ausgerichtet. Zwar tauchen hier ähnlich zur WSDL auch syntaktische Elemente wie z.B. Selektoren in der Beschreibung auf, jedoch lässt die Typisierung der Links Ressourcen miteinander in Beziehungen setzen. In Kombination mit der Semantik der zugeordneten Methode des uniformen Interfaces lassen sich über die Links auch semantische Aussagen über die Interaktionen treffen. Die Kombination aus Link und Protocol stellt ein erweitertes Gegenstück zu den PortTypes der WSDL dar.

ReLL verzichtet im Gegensatz zur WSDL auf eine syntaktische Beschreibung der Ressourcenrepräsentationen, d.h. einzelne Attribute oder Subtypen werden nicht modelliert. Im Kontext von RESTful Services ist dies jedoch auch nicht zwingend erforderlich. So stellt Tilkov in [45] fest, dass diese syntaktischen Informationen maschinenlesbar etwa in XML-Schema-Dokumenten durch die Dienste selbst verwaltet werden können. Hierbei bietet sich der Einsatz der Content-Type Negotiation an.

Ein Äquivalent zu den Ports aus der WSDL existiert in der ReLL nicht. Die Menge der verfügbaren Operationen für einen Link wird in ReLL innerhalb der Protokolle[18] definiert.

[18] vgl. *Protocol* im ReLL-Metamodell

5.1.3 Zweischichtige Kompositionssprache

Pautasso stellt in der Arbeit „Composing RESTful services with JOpera“ [34] mit JOpera eine visuelle Kompositionssprache vor, die sich insbesondere für RESTful Services eignet. Ziel seiner Arbeit ist es u.a., eine Menge von Eigenschaften abzuleiten, die eine Kompositionssprache besitzen muss, wenn sie zur Komposition von RESTful Services geeignet sein soll. Außerdem zeigt er an Hand von JOpera auf, wie diese Anforderungen bei der Umsetzung der Unterstützung von RESTful Services durch die Implementierung neuer Adapter für JOpera umgesetzt werden können.

Da JOpera bereits 2003 als eine generische Kompositionssprache entwickelt wurde (siehe Kapitel 7 in [34]), ist wie bei „BPEL for REST“ auch eine Unterstützung von WS*-Services vorhanden, da diese zu dieser Zeit der aktuellste Stand der Technik waren.

Pautasso stellt heraus, dass das Ziel einer Komposition von RESTful Services nicht die Bündelung des Zustands von fremden Ressourcen ist, sondern dass durch Zustandsübergänge der Kompositionsressource die Zustände der Teilressourcen verändert werden. An dieser Stelle zeigt sich die in Abschnitt 2.2.2 angesprochene „rekursive Natur“ der Dienstkomposition. In Anlehnung an Aristoteles[19] gilt: Das Ergebnis der Komposition ist mehr als die Menge der Teilressourcen.

5.1.3.1 Anforderungen an eine Kompositionssprache

Die REST-Prinzipien lassen sich in Form von Anforderungen auf eine Kompositionssprache projezieren. Abbildung 11 auf Seite 70 visualisiert die Zusammenhänge zwischen den Prinzipien von REST und den Anforderungen an eine Kompositionssprache.

URIs stellen bei REST das zentrale Adressierungskonzept für Ressourcen dar. Die Menge der URIs kann sehr groß werden, da jede Ressource mindestens eine eigene URI besitzt. Außerdem besitzen die URIs eine eigene vom Dienst kontrollierte Dynamik. Das bedeutet, dass von einer URI nicht auf den Typ einer Ressource geschlossen werden darf. Hieraus resultiert die Anforderung der *dynamischen Typisierung*. Ein konsumierter Dienst gibt im Idealfall in der Selbstbeschreibung der Nachrichten den Typ der Ressource und nicht nur nur das Datenformat an. Tilkov stellt hierzu in [45] die Verwendung von Prefixen bei der Beschreibung des Nachrichteninhalts vor (z.B. `application/vnd.mycompany.orderformat+xml`). Ist dies der Fall, sollte eine Kompositionssprache in der Lage sein, den Ressourcentyp bei der Deserialisierung automatisch zu erkennen.

Auf Grund des HATEOAS-Prinzip von REST werden die zu verwendenden URIs innerhalb der Repräsentationen der Ressourcen bekanntgegeben. Eine Kompositionssprache muss daher URIs aus den Verknüpfungen extrahieren können. Zusammen bilden URIs und HATEOAS daher die Anforderung der *dynamischen späten Bindung*, d.h. es entscheidet sich erst zur Laufzeit, welche URIs in den Anfragen verwendet werden.

Da REST kein bestimmtes Format zur Übertragung von Ressourcenrepräsentationen vorschreibt, muss dieses Format mit den Diensten ausgehandelt werden können. Hierzu

[19] Aristoteles sagte einst: „Das Ganze ist mehr als die Summe seiner Teile.“

beschreiben sich die ausgetauschten Nachrichten selbst und geben die gewünschten bzw. gelieferten Datenformate bekannt. Darüber hinaus muss bei der Extraktion von Links das jeweilige Format unterstützt werden. Daher ist der Umgang mit *aushandelbaren Datenformaten* von der Kompositionssprache gefordert.

Die Interaktionen mit den Ressourcen basieren auf einem *uniformen Interface*, so dass durch die Kompositionssprache dessen Umfang nicht eingeschränkt oder dessen Semantik verändert werden darf.

Als eine letzte Anforderung führt Pautasso die *Abrufbarkeit des Kompositionszustands* auf. Ursache hierfür ist die rekursive Eigenschaft der Komposition, nach der sie selbst ebenfalls den Prinzipien von REST folgen muss.

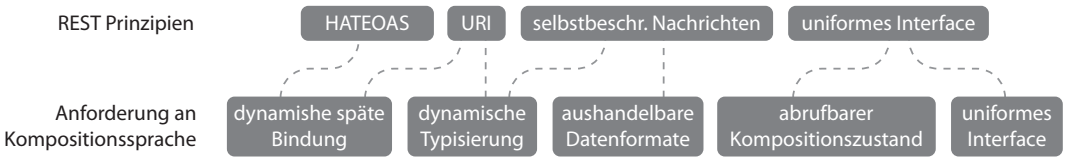

Abbildung 11: Zusammenhänge zwischen den REST-Prinzpien und den Anforderungen an eine REST-konforme Kompositionssprache

5.1.3.2 Kompositionsmodelle in JOpera

Die Modellierung eines Komposition erfolgt visuell in einer Erweiterung der Eclipse IDE. Für Installationshinweise sei an dieser Stelle auf Anhang A.2 hingewiesen. Durch einen visuellen Editor wird es Personen mit geringen Programmierkenntnissen ermöglicht, Kompositionsmodelle zu erstellen. Eine Komposition setzt sich in JOpera aus drei Perspektiven zusammen: *Kontrollflussabhängigkeiten*, *Datenflusstransfer* und *Service Binding*. Abbildung 12 zeigt die Anordnung der einzelnen Abstraktionsschichten eines Prozessmodells in JOpera.

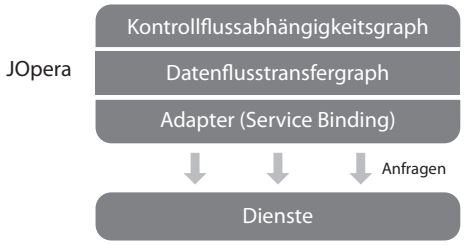

Abbildung 12: Übersicht der Abstraktionsschichten in JOpera

Kontrollflussabhängigkeitsgraph

Ein Geschäftsprozess muss durch den Entwickler, z.B. ein Mitarbeiter der Unternehmensfachseite, in einen gerichteten Graphen abgebildet werden, wobei die Knoten einzelne Aktivitäten und die Kanten auf mögliche folgende Aktivitäten zeigen. Die Abhängigkeiten der

einzelnen Aktivitäten ergeben sich aus der Ausführungssemantik des Modells. Darüber hinaus kann einer Aktivität ein sog. Adapter zugeordnet werden, welcher zur Ausführung von computergesteuerten Programmen wie z.B. Dienstaufrufen eingesetzt werden kann. Hierauf wird bei der Beschreibung des Datenflusstransfers noch näher eingegangen.

JOpera definiert die folgende Ausführungssemantik für einen Kontrollsflussgraph. Wird eine Aktivität beendet, so erzeugt dies ein Ausführungsereignis auf allen abgehenden Kanten. Man kann dies mit dem `fork`-Sprachelement aus BPEL vergleichen. Um nicht auf allen abgehenden Kanten ein Aktivierungsereignis zu versenden, gibt es *selektive Knoten*, die an Hand einer Bedingung entscheiden, welche Kante aktiviert wird. Auf diese Weise wird das `if`-Kontrollkonstrukt, das aus verschiedenen Programmiersprachen bekannt ist, umgesetzt. Von einem Knoten können nicht nur mehrere Kanten abgehen, sondern in ihn können auch mehrere Kanten münden. Im letzteren Fall handelt es sich um einen *Synchronisationsknoten*. In JOpera existieren zwei Synchronisationsstrategien, die auf dem *logischen Und* bzw. *Oder* basieren. Die Synchronisation mittels eines logischen Und aktivieren einen Knoten erst dann, wenn über alle Kanten ein Aktivierungsereignis am jeweiligen Knoten eingegangen ist. Synchronisationsknoten, die ein logisches Oder verwenden, werden im Gegensatz zum logischen Und bei jedem eingehenden Aktivierungsereignis aktiviert[20]. Diese Synchronisationsknoten lassen sich mit dem `join`-Element aus der BPMN vergleichen.

Um Schleifen umzusetzen, muss mit selektiven Knoten und mit Oder-Synchronisationen gearbeitet werden. Dabei stellt die Oder-Synchronisation den Einstieg in die Schleife dar. Das Ende der Schleife bildet ein selektiver Knoten, der entscheidet, ob die schleifenschießende oder die schleifenverlassende Kante aktiviert werden soll.

Datenflusstransfer

Der Datenflusstransfergraph legt fest, wie Daten zwischen den Ein- und Ausgabeparametern der einzelnen Aktivitäten bzw. Adaptern der Komposition ausgetauscht werden. Er stellt eine Sicht auf den Kontrollflussgraph dar, die um technische Details erweitert ist. Daher ist der Datenflusstransfergraph strukturell vom Kontrollflussgraph abhängig. Ziel ist es, den detailierten Datenaustausch von der Ausführungsreihenfolge zu trennen. Die JOpera Dokumentation beschreibt dies, wie folgt:

> *„Abstracting the interface of a service from its access mechanism helps to generalize the scope of the JOpera composition language and system well beyond Web services.“* [26]

Daten werden in Form von Parametern zwischen Aktivitäten sowie Adaptern ausgetauscht. Aktivitätsparameter lassen sich als *fachliche Parameter* verstehen, während Adapterparameter spezifisch für die verwendeten Technologien sind (*technischer Parameter*). Ausgabeparameter können als Wert für mehrere Eingabeparameter verwendet werden. Ein Eingabeparameter hingegen kennt nur eine Quelle für seinen Wert.

[20] Genau betrachtet entspricht die Oder-Synchronisation dem Fehlen von Synchronisation.

Der Datenfluss innerhalb der Komposition kann je nach Szenario eine Zwischenverarbeitung von Daten erfordern. Ein Beispiel hierfür stellt die Extraktion von URIs aus Ressourcenrepräsentationen dar, um die *dynamische späte Bindung* umzusetzen. Diese Zwischenverarbeitungen erfordern jedoch zusätzliche Aktivitäten im Kontrollflussabhängigkeitsgraph. Es kommt somit zu Abhängigkeiten aus einer unteren Abstraktionsschicht an eine darüberliegende.

Mittels XPath- und XSLT-Adaptern lassen sich XML-Dokumente aus Dienstantworten auswerten oder transformieren. Ein XPath-Adapter ist z.B. dafür geeignet, die URI eines bestimmten AtomLinks aus einer Ressourcenrepräsentation zu ermitteln und anschließend als Eingabewert an einen HTTP-Adapter einer nachfolgenden Aktivität weiterzugeben. Dies entkoppelt die Komposition von den URIs der Ressourcen aus den konsumierten Diensten. Allerdings erfordert dies, dass sich der Entwickler der Komposition ggf. in Technologien wie XPath oder XSLT einarbeiten muss, um die Nachrichten korrekt auszuwerten und den Datenflusstransfer korrekt aufzubauen.

Ähnlich zu Pautassos Arbeit „BPEL for REST“ [33] ist es bei der Verwendung von JOpera möglich, URIs aus Templates zu generieren, falls die konsumierten Dienste nicht RESTful sind. Hierfür müssen entsprechende Aktivitäten und Adapter angelegt werden, die Ressourcenrepräsentationen in URIs transformieren.

Setzt man die Adapter aus JOpera mit dem Metamodell von ReLL aus Abschnitt 5.1.2.1 in einen Bezug, so stellen die Adapter die Implementierung der Linkselektoren aus ReLL dar. Ein wesentlicher Unterschied ist jedoch, dass die Adapter so generisch gestaltet sind, dass sie auch für andere Zwecke eingesetzt werden können (z.B. Transformationen der Datenformate).

Service Binding

Pautassos Arbeit zielt besonders auf das Binding von RESTful Services an die einzelnen Aktivitäten der Komposition. Bei seiner Umsetzung verzichtet Pautsso auf eine zusätzliche Beschreibung der konsumierten Dienste, wie z.B. mittels WADL oder ReLL und setzt stattdessen auf eine explizite Modellierung, wie er sie bereits in „BPEL for REST“ verwendet. Das bedeutet, dass der Workflow speziell auf eine Szenario ausgerichtet ist, da er eine Abhängigkeit zu dessen API aufbaut. Zwar bietet sein Ansatz eine Entkopplung von den URIs der verwendeten Ressourcen, jedoch sind die Workflows daran gebunden, dass die verwendeten Datenformate möglichst stabil sind, damit die Zwischenverarbeitung der Interaktionen mit den Diensten nicht „bricht“.

Um RESTful HTTP Services konsumieren zu können, konzipiert Pautasso einen HTTP-Adapter. Dieser kann eine HTTP-Methode, die aufzurufende URI und einen Payload sowie optionale Header-Informationen als technische Parameter entgegennehmen. Die verwendete URI des Adapters kann statisch im Kompositionsmodell kodiert sein oder, wie im vorigen Abschnitt beschrieben, durch URI-Templates oder extrahierte Links bestimmt werden.

Der HTTP-Adapter befolgt darüber hinaus das *uniforme Interface* von HTTP. Die *dynamische Typisierung* wird durch die Header-Informationen der Anfrage bzw. Antwort umge-

setzt. Es ist jedoch die Aufgabe des Entwicklers, diese Informationen bereitzustellen bzw. auszuwerten und zu beachten. Innerhalb der Header-Informationen kann gleichzeitig die *Aushandlung der bevorzugten Datenformate* erfolgen.

5.1.3.3 Komposition als Ressource

In seiner Arbeit [34] geht Pautasso nicht darauf ein, wie ein Prozess als neuer Service veröffentlicht wird. Innerhalb der Dokumentation[21] von JOpera ist jedoch aufgeführt, dass Prozesse automatisch veröffentlicht werden. Dies bedeutet, dass ein in der Eclipse IDE erstelltes Prozessmodell sofort für externe Nutzer verfügbar ist, da das Eclipse Plugin einen eigenen *Application Server* enthält. Prozesse lassen sich optional vor der Öffentlichkeit verbergen, indem in der Eclipse IDE für das jeweilige Prozessmodell das `Published`-Attribut abgeschaltet wird. Die Laufzeitumgebung von JOpera wird über diese Änderung benachrichtigt.

Da JOpera bei der Komposition von Diensten nicht auf eine einzige Architektur von Webservices festgelegt ist, erfolgt bei der Veröffentlichung eines Prozessmodells ebenso keine Festlegung auf eine bestimmte Architektur. Stattdessen werden zu einem Prozessmodell immer eine RESTful HTTP-Schnittstelle und eine SOAP/WSDL-Schnittstelle generiert. Auf diese Weise ist es dem Nutzer der Komposition freigestellt, wie er den Prozess in seine Anwendung integriert.

Um ein Prozessmodell zu instanziieren, muss der Nutzer bei der Verwendung der RESTful HTTP-Schnittstelle eine *POST*-Anfrage an die Prozess-URI des Dienstes senden. Innerhalb der Nachricht müssen vom Konsumenten die zum Start benötigten Parameter als Schlüssel-Wert-Paare mitgesendet werden. JOpera bietet über den Mechanismus der Content-Type-Negotiation für menschliche Konsumenten ein browser-basiertes Frontend, in dem Nutzer die Daten in ein Formular eingeben können. Für maschinelle Konsumenten existiert zusätzlich ein auf XML bzw. JSON basierendes Interface.

5.1.3.4 Trennung von Verantwortlichkeiten

Durch JOpera wird eine Auftrennung der Dienstkomposition in mehrere Modelle mit unterschiedlichen Schwerpunkten angestrebt. Die saubere Trennung von technischen Details und dem fachlichen Prozess gelingt jedoch nicht immer. Die Verwendung von eigenen Aktivitäten zur Zwischenverarbeitung von Daten führt dazu, dass der Kontrollfluss nicht komplett vom Datenflusstransfer entkoppelt betrachtet werden kann, d.h. die Struktur des Prozesses korreliert mit den zugrundeliegenden Technologien (z.B. Extraktion von URIs für nachfolgende Aufrufe).

Pautasso merkt jedoch an, dass die Softwarewerkzeuge zur Erstellung der Workflows in der Lage sind, eine Synchronisation zwischen Datenflussmodell und Kontrollflussabhängigsmodell herzustellen. Damit wird die Schwere dieses Schwachpunkts reduziert.

[21] http://www.jopera.org/docs/help/jop_8.html#sec_wsdl

5.2 Umgang der Ansätze mit der Problemstellung

In Kapitel 1.2 sind mehrere Teilprobleme aufgelistet, die durch den Einsatz von RESTful Services in der Anwendungsentwicklung mit Schwerpunkt auf dienstorientierte Architekturen im Unternehmensumfeld entstehen. Nun sollen in diesem Abschnitt die vorgestellten Arbeiten von Pautasso et al. und Alarcon et al. in einen Zusammenhang zu diesen Problemen gebracht werden. Dabei werden die jeweiligen Ziele, Stärken und Schwächen der Arbeiten betrachtet, so dass eine Empfehlung für weitere Entwicklungen gegeben werden kann.

5.2.1 Schnittstellenbeschreibung

In der Arbeit „BPEL for REST“ [33] orchestriert Pautasso mehrere RESTful HTTP Services innerhalb eines BPEL-Prozessmodells. Da er im Gegensatz zu WS*-Services nicht auf eine maschinenlesbare Beschreibung des Dienstes zugreifen kann, werden von ihm sämtliche Informationen, die für die Ausführung des Prozesses notwendig sind, innerhalb des Prozessmodells festgelegt. Dies bedeutet, dass der Entwickler des Prozesses (im Idealfall die Unternehmensfachseite) sich mit dem HTTP und Datenformaten auseinandersetzen muss, um den Prozess korrekt zu modellieren. So muss der Entwickler die Semantik der HTTP Methoden verstehen und korrekt anwenden. Außerdem muss er die Statuscodes des HTTP interpretieren und mit korrekten Codes antworten können, da sie über das Ergebnis eines Dienstaufrufs Aufschluss geben. Der Entwickler muss zusätzlich verarbeitende Technologien zu den jeweilig verwendeten Datenformaten kennen und anwenden können, um z.B. mittels XPath Teile einer XML-Repräsentation zu extrahieren und anschließend in den Payload einer Anfrage einzusetzen. Somit gilt für „BPEL for REST“, dass es sich im Hinblick auf die fehlende Schnittstellenbeschreibung von RESTful Services um eine sehr technisch orientierte Lösung handelt, die auf eine Abstraktionsschicht zwischen den Diensten und dem jeweiligen Prozessmodell verzichtet.

Pautasso liefert mit JOpera [34] einen alternativen Umgang mit der fehlenden Schnittstellenbeschreibung. Zwar werden die RESTful Services nach wie vor nicht durch ein zusätzliches Dokument beschrieben, jedoch werden die technischen Details der Interaktion mit den Diensten in eine eigene Abstraktionsschicht (Datenflusstransfermodell) verschoben. Das Datenflusstransfermodell enthält zwar ähnlich zu „BPEL for REST“ sämtliche Detailinformationen über den Umgang mit den verwendeten Ressourcen, jedoch sind diese Details im Idealfall nicht im Kontrollflussabhängigkeitsgraph des Prozesses sichtbar. Da die ausgetauschten Nachrichten in eigenen Aktivitäten verarbeitet werden müssen, kommt es zu Abhängigkeiten vom Datenflusstransfermodell zum Kontrollflussabhängigsgraph. Somit ist die Effektivität der Trennung von abstrakten Aktivitäten und konkreten Dienstaufrufen nicht ideal. Die Idee der Verwendung einer zusätzlichen Abstraktionsschicht bzw. eines eigenen Modells für Ausführung der Interaktion mit den RESTful Services ist jedoch als sinnvoll zu bewerten. Eine möglicher Ansatz zur Eliminierung der Abhängigkeiten zwischen Datenflusstransfer und Kontrollfluss könnte die Bindung von mehreren geordneten

Adaptern an eine einzelne Aktivität darstellen, da somit auf zusätzliche technologisch begründete Aktivitäten verzichtet werden könnte. So kommt es z.B. vor der Durchführung einer HTTP-Anfrage idR. zur Extraktion der zu verwendenden URI. Die beiden hierzu benötigten Adapter könnten in einer Aktivität nacheinander ausgeführt werden, ohne dabei einen strukturellen Einfluss auf den Kontrollflussabhängigkeitsgraph zu haben.

Alarcon et al. schlagen im Vergleich zu den Ansätzen von Pautasso im Umgang mit der fehlenden Schnittstellenbeschreibung von RESTful Services eine weitere alternative Richtung ein. Sie stellen mit ReLL ein Metamodell vor, welches zur Beschreibung von Ressourcen und deren Beziehungen zueinander geeignet und von einer Prozessengine unabhängig ist. Somit wird eine Dienstbeschreibung nicht als Menge von Operationen mit bestimmten Typen von Parametern und Ausgabewerten verstanden, sondern als der Zusammenhang der Ressourcen zueinander. Damit enthält ein ReLL-Modell Informationen über die möglichen Interaktionsabläufe und Zustandsübergänge von Ressourcen. Ein solches Ressourcenbeschreibungsmodell ermöglicht es in einer Prozessmodellierung, die Interaktion mit einem Dienst als das Folgen von bereits modellierten Verknüpfungen zwischen den Ressourcen zu verstehen. Die Entdeckung und Beschreibung der Verknüpfungen wird somit nicht erst im Prozessmodell formuliert, wie es etwa bei „BPEL for REST“ [33] oder JOpera [34] der Fall ist.

5.2.2 Dienstsemantik

In Pautassos Arbeit „BPEL for REST“ wird die Dienstsemantik während der Entwicklung des Prozessmodells geklärt. Der Entwickler des Prozesses muss sowohl die Bedeutung des abstrakten Geschäftsprozesses als auch die der verwendeten Dienste kennen, um ein valides Prozessmodell zu erstellen. Die Sprache der Dienste unterscheidet sich durch die Verwendung von RESTful HTTP von der der operationsorientierten Dienste. So muss der Entwickler des Prozessmodells die Bedeutung des uniformen Interfaces, der Ressourcen und der Metadaten aus den Nachrichten verstehen. Daher erfordert eine Umsetzung von „BPEL for REST“ in der Praxis, dass die ausführbaren BPEL-Prozesse nicht mehr von einem einzelnen Entwickler erstellt werden, sondern dass diese mittels einer kollaborativen Methodik durch Unternehmensfachseite und IT-Abteilung zusammen modelliert werden. Im Dialog zwischen den beiden Seiten findet dabei die Abbildung von abstrakten Geschäftsprozessen auf die Operationen und Ressourcen der Dienste statt. Im Vergleich zum Vorgehen aus WS*-Architekturen ergibt sich somit ein höherer Kommunikationsaufwand. Die spontane Änderbarkeit der Prozessmodelle wird zusätzlich eingeschränkt, da die Konsequenzen von Änderungen an den Prozessmodellen nicht durch einen Entwickler alleine abgeschätzt werden können.

Die Auftrennung der JOpera Komposition in mehrere spezialisierte Abstraktionsschichten setzt an der Problematik der Kommunikation zwischen Fachseite und IT, wie sie bei „BPEL for REST“ existiert, an. Zwar kann ebenfalls nicht ein Entwickler allein das gesamte Prozessmodell erstellen, jedoch bieten die einzelnen Teilmodelle für beide Seiten der Ent-

wicklung eine unterschiedlich detaillierte Betrachtung des Prozesses. Die Unternehmensfachseite kann mittels des Kontrollflussabhängigkeitsgraphen die Abfolge der Aktivitäten des Prozesses innerhalb einer visuellen Notation unter Berücksichtigung der jeweiligen Aktivitätssemantik modellieren. Dieses Modell stellt den Startpunkt für einen IT-Entwickler dar, der im Datenflusstransfergraph beschreibt, wie die fachlichen und technischen Parameter von den Adaptern verwendet werden müssen. Damit stellt die Erstellung des Datenflusstransfergraphs den Schritt dar, in dem die Bedeutung der fachlichen Aktivitäten und Objekte für den IT-Entwickler geklärt werden müssen. Ähnlich zu „BPEL for REST“ muss dies in einem Dialog zwischen Fachseite und IT erfolgen.

Damit unterscheiden sich „BPEL for REST“ und JOpera in ihrem Umgang mit der Klärung der Dienstsemantik nicht stark, jedoch stellt JOpera auf Grund der visuellen Notation eine geeignetere Kommunikationsgrundlage dar, die den Dialog zwischen Fachseite und IT-Entwicklern erleichtert.

Die von Alarcon et al. vorgestellte ReLL ermöglicht eine menschenlesbare und maschinenlesbare Beschreibung der Zusammenhänge von Ressourcen. Die menschenlesbare Beschreibung ist innerhalb des ReLL Metamodells für Dienste, Ressourcen, deren Repräsentationen und Link-Typen in Form eines Freitextattributs vorgesehen. Die maschinenlesbare Beschreibung macht die technische Bedeutung der Zusammenhänge zwischen den Ressourcen durch die Links explizit. REST legt nicht fest, dass URIs mit einer festen Semantik gekennzeichnet werden müssen:

> *„Semantics are a by-product of the act of assigning resource identifiers and populating those resources with representations. At no time whatsoever do the server or client software need to know or understand the meaning of a URI — they merely act as a conduit through which the creator of a resource (a human naming authority) can associate representations with the semantics identified by the URI.“* [14] Abs. 6.2.4

Links haben lediglich die Aufgabe, mögliche Aktionen mit einer Ressource aufzuzeigen. Soll ein Geschäftsprozess mittels RESTful Services umgesetzt werden, so ist es die Aufgabe für den Entwickler, im Prozessmodell den richtigen Links zu folgen, um die gewünschten Zustandsänderungen der Ressourcen zu erreichen. Damit dies möglich ist, muss zum einen die Bedeutung der Links bekannt und zum anderen die zu verwendende URI vorhanden sein. Zur Beschreibung der Bedeutung kann die menschenlesbare Beschreibung der Link-Typen verwendet werden. Die Methoden des uniformen Interfaces lassen sich im ReLL-Protokoll eines Links hinterlegen, so dass der Entwickler des Prozesses nicht notwendigerweise die Semantiken des uniformen Interfaces kennen muss.

Eine tatsächliche Anfrage an einen Dienst besitzt z.B. beim Erstellen oder Aktualisieren einer Ressource deren Repräsentation als Payload. Das Vorgehen zur Erzeugung dieses Payloads ist nicht durch ReLL abgedeckt, da das ReLL-Metamodell hierfür nicht konzipiert ist. Die Serialisierung des Payloads ist wie bei der Verwendung von WSDL Aufgabe der Ausführungsumgebung des Prozesses. Im Prozessmodell muss definiert werden, welche Teile

der ausgetauschten Nachrichten erstellt, wiederverwendet, geändert oder verworfen werden müssen. Ein Objekt aus der Anwendungsdomäne muss auf ein konkretes Repräsentationsformat abgebildet werden. Dieser Abbildungsschritt setzt ein Wissen über die Anwendungsdomäne als auch über die Implementierung des Dienstes voraus.

Da die Semantik der Methoden des uniformen Interfaces fest definiert ist, müssen die Bedeutungen der Aktivitäten der Anwendungsdomäne auf Ressourcen abgebildet werden. „BPEL for REST“ und „JOpera“ setzen voraus, dass bei der Entwicklung eines Prozessmodells Wissen über die Semantik der verwendeten Ressourcen vorhanden ist, da die Zuordnung von Aktivitäten zu Dienstaufrufen manuell durchgeführt werden muss. Zusammenhänge zwischen den Ressourcen müssen im Prozessmodell definiert werden. Dadurch ist die Bedeutung der Verknüpfungen nur implizit im Modell festgehalten. ReLL modelliert die Zusammenhänge der Ressourcen explizit. Ressourcen spiegeln Konzepte der Anwendungsdomäne wieder und werden mit weiteren Ressourcen in Bezug gesetzt. Daraus ergibt sich ein Verständnis über die Strukturen der verwendeten Ressourcen, welches bei „BPEL for REST“ oder JOpera informell verwaltet wird.

Somit bietet ReLL bei der Entwicklung von Prozessen mehr Unterstützung bei der Definition und dem Verständnis der Dienstsemantiken, als es ohne eine solche Beschreibungssprache etwa bei „BPEL for REST“ oder JOpera der Fall ist.

5.2.3 Umgang mit Hypermedia

Sowohl „BPEL for REST“ als auch JOpera unterstützen die Nutzung von Hypermedia als Clients der konsumierten Dienste. In beiden Ansätzen stellt die Verarbeitung der Links einen expliziten Teil des modellierten Prozesses dar. D.h. der Prozessentwickler muss bewusst entscheiden, welche Verknüpfungen er benötigt und verwenden muss.

Betrachtet man die Umsetzungen, so werden jedoch Unterschiede zwischen den beiden Ansätzen deutlich. Pautasso setzt in „BPEL for REST“ voraus, dass die konsumierten Dienste die URIs der Verknüpfungen als Attribute innerhalb der Ressourcenrepräsentationen ablegen. Er geht somit nicht von einem standardisierten Format wie etwa Atom aus, sondern beschränkt sich auf die Implementierung, die ihm vom Dienst seines Fallbeispiels angeboten wird. Eine Verwendung von AtomLinks wäre innerhalb seines Modells umsetzbar, jedoch würde dies die Verarbeitung der Nachrichten komplexer gestalten.

JOpera hingegen weist explizit auf die Verwendung von spezifischen Adaptern hin wie z.B. zur XML-Verarbeitung mittels XPath geeignet sind. Somit kann der Entwickler des Prozessmodells auf Werkzeuge zurückgreifen, die es ermöglichen mittels spezieller Ausdrücke (z.B. XPath-Ausdrücke) die benötigten URIs aus einer Ressourcenrepräsentation zu extrahieren. Die dafür notwendigen Verarbeitungsschritte werden im Datenflusstransfergraph modelliert, jedoch müssen u.U. auch Änderungen an der Struktur des Kontrollflussabhängigkeitsgraphen vorgenommen werden.

Alarcon et al. stellen mit der ReLL eine Sprache vor, die für die Beschreibung der Ressourcenzusammenhängen ausgelegt ist. Daher werden Links nicht auf eine URI mit einem

zusätzlichen Bezeichner reduziert, sondern um Typ- und Protokollinformationen erweitert. Darüber hinaus stellen die Selektoren eine von einer Prozessengine losgelöste Beschreibung zur Extraktion der Links dar. Dadurch dass sich ReLL nicht an eine konkrete Prozessengine oder Prozessbeschreibungssprache bindet, hängt es von diesen ab, wie effektiv die Informationen eines ReLL-Modells genutzt werden. Alarcon et al. zeigen mittels eines Petri-Netzes, dass sich ReLL dafür eignet Verarbeitungsschritte, die sich auf die Verarbeitung von Hypermediainformationen beziehen, im Prozessmodell auf ein Minimum zu reduzieren.

5.2.4 Integration mit WS*-Architekturen

Vor dem Hintergrund der Integration von RESTful Services mit WSDL und SOAP-basierten Services innerhalb eines Prozessmodells stellt Pautasso mit „BPEL for REST“ eine Lösung vor, die auf den bisherigen Stand der WS*-Technologien aufbaut. Dadurch ist es möglich bereits entwickelte BPEL-Prozessmodelle so zu erweitern, dass auch RESTful Services Bestandteil einer Komposition sein können bzw. aus einer aktiven Prozessinstanz Ressourcen veröffentlicht werden können.

JOpera stellt eine visuelle Kompositionssprache dar, die ursprünglich zur Komposition von WS*-Services konzipiert ist. Da die Kontrollflussabhängigkeitsgraphen dienstunabhängig sind und erst die Verknüpfung der Aktivitäten mit architekturspezifischen Adaptern eine Bindung der Komposition an konkrete Dienste darstellt, lässt sich in JOpera-Prozessmodellen eine Vielzahl von heterogenen Diensten verwenden. Die Ausführung von JOpera-Prozessmodellen ist an die JOpera-Prozessengine gebunden, so dass ggf. bereits vorhandene BPEL-Prozessengines nicht weiterverwendet werden können.

Die ReLL bietet keine Unterstützung für SOAP-basierte Dienste, da diese bewusst nicht Teil der Zielsetzung von ReLL sind. Eine Integration von WS*- mit RESTful Services kann jedoch in Abstraktionsschichten oberhalb der Beschreibung der Ressourcenzusammenhänge erfolgen. So ist es denkbar, dass eine hybride Prozessengine auf das ReLL-Modell eines RESTful Services und die WSDL-Beschreibung eines WS*-Services zugreift, um diese beiden architektonisch unterschiedlichen Dienste innerhalb eines Prozesses individuell zu steuern und dadurch zu integrieren.

5.3 Anwendung am Fallbeispiel

Auf Grund der Unterstützung sowohl von WS*- als auch RESTful HTTP Services stellt JOpera eine Option für die Implementierung des Fallbeispiels aus Kapitel 3 dar. Darüber hinaus bietet die Trennung von Ablaufabhängigkeiten der Aktivitäten vom Datenfluss zwischen den Aktivitäten unterschiedliche Abstraktionen für ein Prozessmodell an, so dass bei der Entwicklung inkrementell vorgegangen werden kann. „BPEL for REST“ hingegen führt sämtliche Informationen in einer einzelnen Abstraktionsschicht zusammen. Eine Umsetzung mittels „BPEL for REST“ ist außerdem nicht möglich gewesen, da die hierzu benötigten Werkzeuge nicht auffindbar sind. Daher soll nun das Fallbeispiel als JOpera-Prozess modelliert werden, um die Praxistauglichkeit von JOpera bewerten zu können.

Das Kontrollflussabhängigkeitsmodell beginnt mit mehreren Aktivitäten. Die wichtigste ist hierbei der Aufruf der verfügbaren Dienste über den Einstiegspunkt des Contract and Customer Service. Aus der Repräsentation muss in einer nächsten Aktivität die URI der Vertragslistenressource extrahiert werden, anschließend kann ein leerer Vertrag erstellt werden. Sobald der Vertrag erstellt ist, kann die Bonität des Kunden geprüft werden. Hierzu muss ein zusätzlicher Dienst (Rating-Service aus Abs. 3.2) über eine SOAP-Schnittstelle aufgerufen werden. Damit dieser Aufruf durchgeführt werden kann, muss in mehreren Schritten das Anfrageobjekt erstellt werden. Aus der Antwort des Rating-Dienstes wird der relevante Wert extrahiert, um anschließend eine von zwei exklusiven Aktivitäten durchzuführen, die den Status des Vertrags je nach Bonitätsbewertung auf kreditwürdig oder nicht-kreditwürdig setzt.

Zu Beginn der Prozessausführung werden neben dem eben beschriebenen Ablauf gleichzeitig die XML-Repräsentationen für die Bankdaten und Kundendaten erstellt, da diese keine logischen Abhängigkeiten auf andere Aktivitäten besitzen.

Sobald sämtliche Vertragsinformationen gesammelt sind, kann die XML-Repräsentation aus den einzelnen Fragmenten zusammengesetzt werden. Zusammen mit der URI des erstellten Vertrags, werden dessen Informationen in einer weiteren Aktivität aktualisiert. Die URI ist hierbei aus der initialen Repräsentation des Vertrags nach dessen Erstellung entnommen. Nachdem die Vertragsinformationen aktualisiert sind, kann zum einen die URI zur Vertragsaktivierung extrahiert und zum anderen die Repräsentation der Vertragsaktivierung erstellt werden. Sollte kein Link zur Vertragsaktivierung gefunden werden, so wird die „activateContract“-Aktivität übersprungen und der Vertrag nicht aktiviert.

Diese Anwendung von JOpera auf das Fallbeispiel zeigt, wie das HATEOAS-Prinzip in einer Kompositionssprache umgesetzt werden kann. Darüber hinaus wird die Schwäche von JOpera deutlich, dass technische Details Einfluss auf das Modell des Kontrollflusses nehmen. So erfordert die Extraktion von URIs oder das Erstellen von Repräsentationen zusätzliche Aktivitäten. Dies gilt im übrigen bei JOpera sowohl für RESTful HTTP als auch WS*-Services. An dieser Stelle sei außerdem auf ein wiederkehrendes Muster hingewiesen: Vor einer Aktivität, an die ein HTTP-Adapter gebunden ist, steht idR. ein Tupel aus Aktivitäten zur Link-Extraktion und zum Repräsentationszusammenbau. Diese beiden vorgeschalteten Aktivitäten können parallel verarbeitet werden und werden vor dem Dienstaufruf synchronisiert. Abbildung 13 zeigt den erstellten Kontrollflussabhängigkeitsgraph, damit die Erläuterungen nachvollzogen werden können.

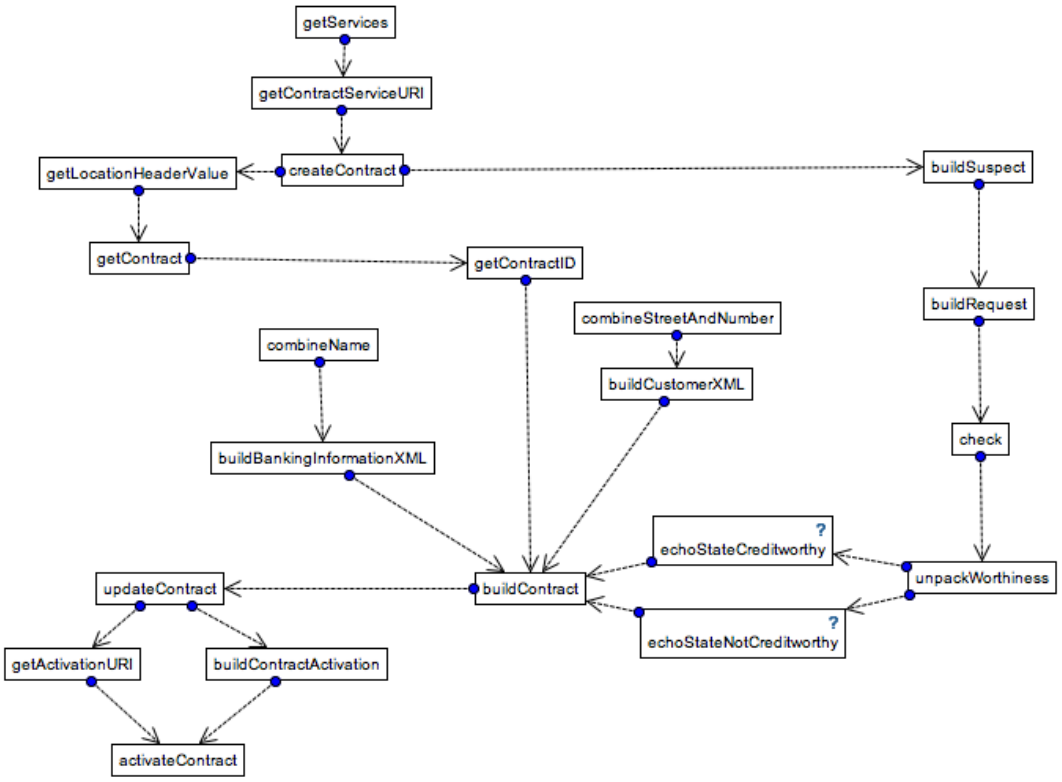

Abbildung 13: Kontrollflussabhängigkeitsgraph angewandt auf das Fallbeispiel aus Abs. 3

Bei der Umsetzung des Datenflusstransfergraphen existiert eine pragmatische, technische Hürde: Die aktuelle Version 2.5.4 von JOpera besitzt Implementierungsfehler innerhalb des XPath-Adapters, die das Auslesen einzelner Werte aus Attributen verhindern. Daher muss auf die Vorgängerversion 2.5.3 zurückgegriffen werden, um das Beispiel erfolgreich ausführbar machen zu können[22].

Der Datenflusstransfergraph ist in Abbildung 14 dargestellt. Hier ist zu sehen, dass die Prozesseingabeparameter direkt zum Prozessstart an die Parameter der repräsentationserzeugenden Adapter (`buildSuspect`, `buildContract` etc.) weitergereicht werden. Ebenfalls hervorzuheben ist, dass direkt nach dem Start des Prozesses die Verarbeitung der Service-URIs erfolgt (`getServices` und `getContractServiceURI`). Das Ergebnis des Prozesses ist der HTTP-Statuscode der Vertragsaktivierung. Im Erfolgsfall liefert der Prozess daher das Ergebnis „200“.

Im Anhang A.2 befindet sich eine Anleitung, wie der Prozess über die Kommandozeile instanziiert werden kann. Außerdem befindet sich dort eine Anleitung zur Installation von JOpera und dem hier beschrieben Prozessmodell.

[22] siehe Foreneintrag http://www.jopera.org/node/482

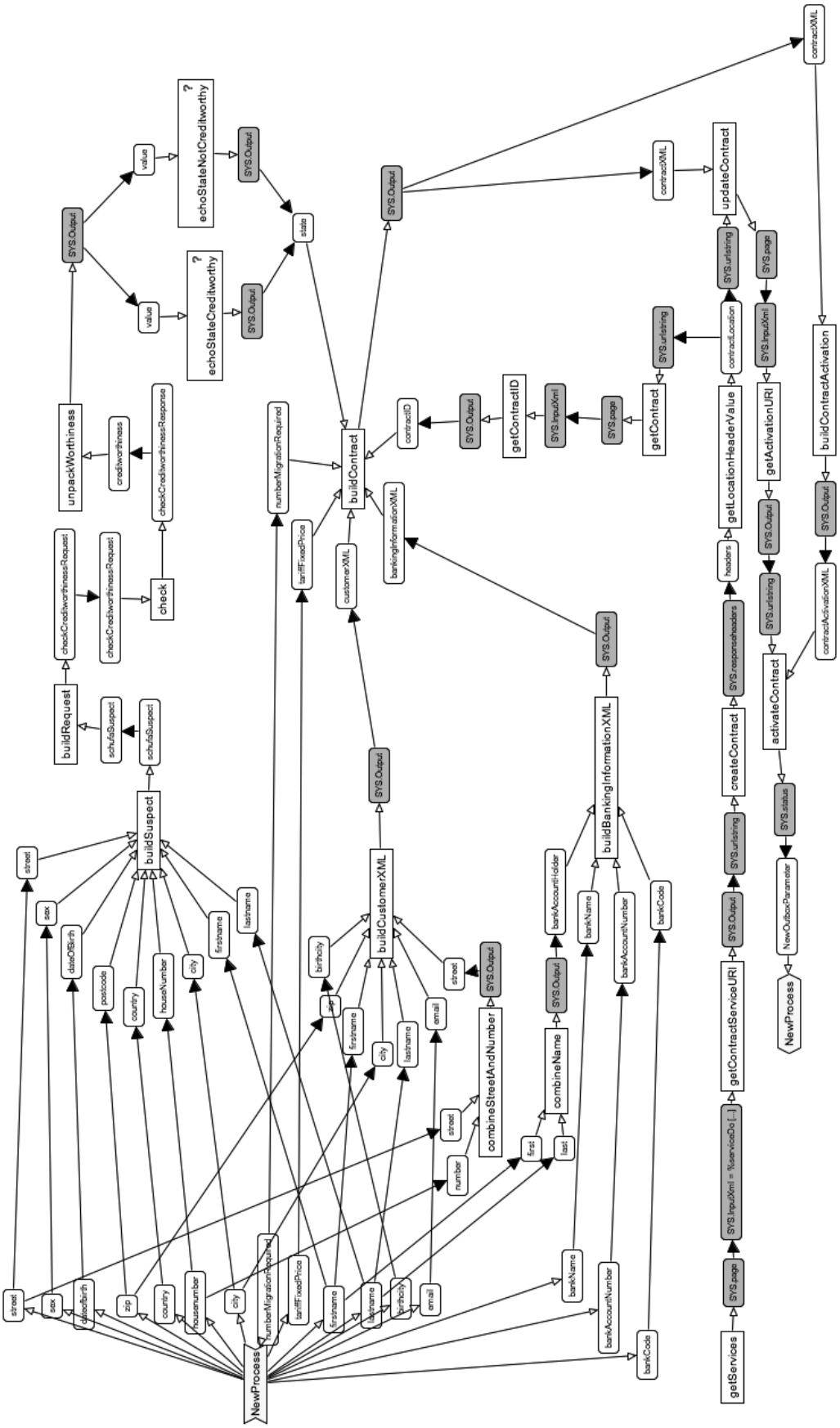

Abbildung 14: Datenflusstransfergraph angewandt auf das Fallbeispiel aus Abs. 3

5.4 Einsatzempfehlungen der Ansätze

Aus den Charakteristiken der in Abschnitt 5.2 beschriebenen Ansätze zur Komposition von RESTful Services sowie der Anwendung von JOpera auf ein eigenes Fallbeispiel leiten sich unterschiedliche Empfehlungen ab, für welche Einsatzszenarien die jeweiligen Ansätze geeignet sind.

Die Erweiterung „BPEL for REST“ stellt eine Lösung für die Integration von RESTful Services in BPEL-Prozessmodelle dar. Damit eignet sich dieser Ansatz besonders für Szenarien, in denen bereits BPEL-Prozessmodelle existieren, die nun neben den SOAP-basierten Diensten auch RESTful Services konsumieren müssen. Grund hierfür kann unter anderem die Einstellung der Unterstützung einer API durch einen externen Dienstanbieter sein. Da die Interaktion mit den RESTful Services innerhalb des Prozessmodells gesteuert werden muss, ist es notwendig, dass sich der Entwickler des Prozessmodells mit den Eigenschaften von RESTful Services auseinandersetzt und den Prozess RESTful[23] gestaltet.

Sollte es aus unternehmerischer Perspektive (z.B. finanziell oder Einhaltung von Standards) möglich sein, die existierenden Prozessmodelle in eine andere Sprache zu übersetzen, so bietet sich der Einsatz von JOpera an. Zwar müssen sich auch hier die Entwickler der Prozessmodelle mit technischen Details auseinandersetzen, um einen vollständig funktionalen, ausführbaren Prozess zur erhalten, jedoch können die Prozesse in unterschiedlichen Detailgraden entworfen werden. Dies bietet den Vorteil, dass fehlendes Wissen über RESTful Services durch die Hinzunahme von IT-Personal bei der Erstellung der Datenflusstransfergraphen ausgeglichen werden kann. Dieses Vorgehen setzt voraus, dass die Kommunikation zwischen der Unternehmensfachseite und dem IT-Personal effektiv ist und die jeweiligen Bedeutungen von Aktivitäten sowie Ressourcen und deren Veknüpfungen geklärt werden.

Für den Einsatz der ReLL ist es schwierig eine Empfehlung abzugeben, da es sich nur um eine Beschreibungssprache für der Hypermediabestandteile von Ressourcen handelt. Daher hängt der Einsatz vorrangig von der Unterstützung durch Prozessmodellierungssprachen und zugehörige Prozessengines ab. Die Verwendung von Petrinetzen als Steuerungseinheit von Prozessen wie es von Alarcon et al. demonstriert wird ist zu hinterfragen, da Petri-Netze sehr generisch und in der Praxis eher weniger verbreitet sind. Neben den Eigenschaften von REST verursachen sie nochmals zusätzlichen Mehraufwand beim Einarbeiten der Entwickler.

Für die ReLL lässt sich jedoch eine umfassendere Empfehlung an die Entwickler von Prozessmodellierungssprachen und Prozessengines ableiten. Diese sollten sich darum bemühen Links innerhalb der Ressourcenrepräsentationen durch eine eigene Abstraktionsschicht in der Prozessmodellierung - z.B. durch ReLL - für die Entwickler der Prozessmodelle verfügbar zu machen. Dadurch können die Prozessmodelle weiter von technischen Details wie z.B. dem Parsing von Repräsentationen, befreit werden. Außerdem lassen sich derartige Beschreibungen innerhalb verschiedener Prozessmodelle auch über die Organsiationsgrenzen hinaus wieder verwenden.

[23] d.h. unter Einhaltung der Prinzipien des Architekturstils REST

5.5 Spannungsverhältnis zwischen Dienstdynamik und Prozessmodellen

Jeder der beschriebenen Ansätze basiert auf Annahmen darüber, wie ein Dienst implementiert ist. Dies zeigt sich bei der Umsetzung des HATEOAS-Prinzips. Zwar können hier Ressourcen ohne URI-Templates adressiert werden, jedoch muss ein Client Annahmen darüber treffen, dass eine Ressourcenrepräsentation abhängig vom Zustand der Ressource bestimmte Links enthält bzw. sich hinter einem Link ein bestimmter Ressourcentyp verbirgt. Diese Form der Kopplung kann dadurch abgeschwächt werden, dass die Dienste die ausgetauschen Nachrichten mit versionierten dienstspezifischen Formattypen versehen. So kann sichergestellt werden, dass eine Ressourcenrepräsentation einen bestimmten Typ besitzt und damit über für dieses Format definierte Links verfügt. Statt eines versionierten Datenformats ließe sich auch ein versionierter Einstiegspunkt in den Dienst verwenden, der dann die Ressourcenrepräsentationen innerhalb eines zur Version passenden Formats ausliefert. Auf diese Weise kann ein Dienst dynamischer gestaltet werden.

Die Verarbeitung von Repräsentationen zur Extraktion der Links führt zu einem höheren Anteil von technischen Details in den Prozessmodellen. Dies zeigt sich sowohl in „BPEL for REST“ als auch in JOpera. Die Verwendung von URI-Templates kann zu einer Verringerung der technischen Detailanteile beitragen, da die Abstraktion der URIs durch Links fehlt. Allerdings werden dadurch die Clients gleichzeitig stärker an den Dienst gekoppelt und nicht mehr durch Hypermedia gesteuert.

Es zeichnet sich demnach ein Konflikt zwischen der Dynamik der Dienste bzw. der losen Kopplung der Clients an die Dienste und der Technologiebefreiung der Prozessmodelle ab. Durch unterschiedliche Maßnahmen wie etwa die Versionierung der Datenformate oder eine zusätzliche Abstraktionsschicht zur Beschreibung der Verknüpfungen zwischen den Ressourcen können den Konflikt abschwächen.

Unabhängig vom konkreten Ansatz, muss bei der Komposition von RESTful Services ein Kompromiss zwischen der Kopplung der Clients an die Dienste sowie der Separierung der Verantwortlichkeiten zwischen Fachseite und IT gefunden werden.

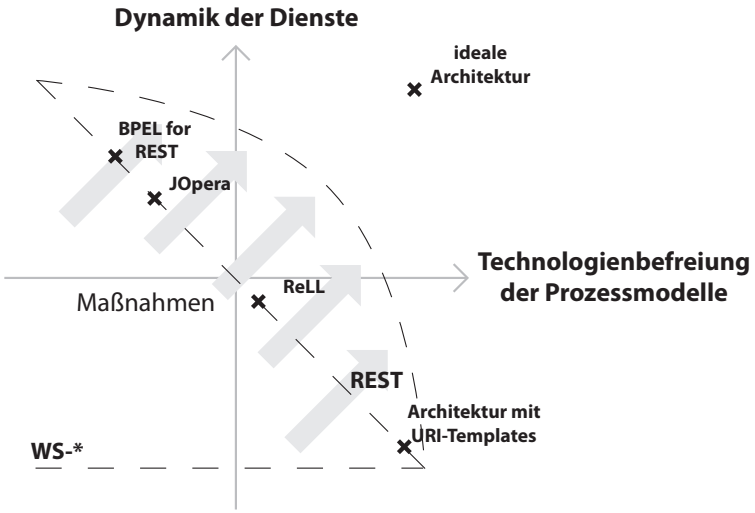

Abbildung 15: Konflikt zwischen Dienstdynamik und technischen Details in Prozessmodellen

6 Service Discovery

Im vorangehenden Kapitel ist gezeigt worden, dass durch die Einräumung von Dynamik für RESTful Services die Bindung dieser Dienste an Aktivitäten eines Prozessmodells technisch komplexer wird. Dieses Spannungsverhältnis leitet daher zur Thematik der Service Discovery über. Es soll gezeigt werden, dass das Problem der Auffindung von Diensten durch die Verwendung von REST zwar nach wie vor existiert, jedoch an einer anderen Stelle des Entwicklungsprozesses einer Anwendung bearbeitet wird.

6.1 Ziel

Ziel der Service Discovery ist es, an Hand von Metadaten zur Entwicklungs- oder Laufzeit Dienste zu entdecken, die eine gewünschte Aufgabe lösen können (siehe Tilkov [45] S. 189). Mit einer automatisierten dynamischen Entdeckung von Diensten wird einer syntaktischen Kopplung (siehe Abs. 2.1.1) entgegengewirkt.

In der Definition des SOA-Begriffs in Abschnitt 2.2.1 ist herausgestellt worden, dass es in einer dienstorientierten Architektur u.a. darum geht, die von der IT bereitgestellten Dienste nach den Bedürfnissen des Unternehmens auszurichten. Bei der Implementierung einer SOA stellt die *semantische Lücke* zwischen den Sprachen der Unternehmensfachseite und der IT ein zentrales Problem dar. Die semantische Lücke wird dadurch charakterisiert, dass unterschiedliche Konzepte der Anwendungsdomäne (z.B. Prozesse, Aktivitäten oder Geschäftsobjekte) auf Konzepte der Softwareentwicklung (z.B. Operationen, Datenstrukturen oder Ressourcen) treffen. Somit kommt die Notwendigkeit für eine Übersetzung der unterschiedlichen Sprachen zustande. Welche Funktionalitäten der Dienste dafür geeignet sind, die einzelnen Aktivitäten des Prozesses durchzuführen, hängt von der Bedeutung der jeweiligen Aktivitäten und Dienstfunktionalitäten ab. Um die Selektion der Funktionalitäten durchzuführen, müssen die Konzepte aus der Anwendungsdomäne auf Konzepte der Softwareentwicklung übertragen und validiert werden. Dieser Vorgang kann der Service Discovery zugeordnet werden, da Metadaten auch in Form von informellen Gesprächen während der Entwicklungszeit ausgetauscht werden können.

Im Forschungsbereich der Verarbeitung von Bilddaten wird der Begriff der semantischen Lücke dazu verwendet, um die *„Lücke zwischen der computerbasierten Repräsentation und der menschlichen Interpretation des Bildes“* (übersetzt aus [12]) zu benennen. In Anlehnung an Smeulders et al. [43] soll der Begriff der semantischen Lücke nun im Kontext dienstorientierter Architekturen wie folgt definiert werden:

> *Die semantische Lücke ist das Fehlen eines Zusammenhangs zwischen den Informationen einer ausführbaren und voll funktionsfähigen Prozessbeschreibung und der Interpretation eines informellen Prozessmodells.*

Das bedeutet, dass bei der Erstellung eines ausführbaren Prozessmodells die Zusammenhänge zwischen den Diensten und dem Wissen über die Anwendungsdomäne durch den Entwickler erkannt werden müssen. Im Fall von REST-Architekturen gilt, dass die Erstellung

des Kompositionsmodells für die Überbrückung der semantischen Lücke verantwortlich ist; Grund hierfür ist, dass die erstellte Komposition die unterschiedlichen Konzepte vereint und dadurch die Zusammenhänge zwischen der Anwendungsdomäne und den Diensten impliziert.

Damit die semantische Lücke geschlossen werden kann, muss ein Dialog mittels Methoden der Softwareentwicklung zwischen der Unternehmensfachseite und den Dienstanbietern bzw. -entwicklern stattfinden. Zusätzlich kann die Dokumentation der Dienstschnittstellen für die Fachseite eine Entscheidungshilfe darstellen, um bei der Entwicklung eines Prozessmodells den Aktivitäten die richtigen Funktionalitäten der Dienste zuzuweisen.

Im Folgenden soll nun betrachtet werden, wie sich der Entwicklungsprozess einer Dienstkomposition durch REST verändert und an welcher Stelle der Entwicklung die semantische Lücke geschlossen wird.

6.2 Semantische Lücke

6.2.1 Auswirkungen des WS*-Stil

In einer WS*-Architektur stellt die WSDL-Beschreibung eines Dienstes zwar nur syntaktische Interoperabilität her, jedoch ist die WSDL-Beschreibung in „contract first“-Sezenarien zusätzlich das Ergebnis einer semantischen Synchronisation zwischen IT und Unternehmensfachseite. In einem solchen Szenario handeln Dienstkonsument und Dienstanbieter einen statischen Vertrag aus, von dem sie nicht mehr ohne Absprache mit dem jeweiligen anderen Partner abweichen werden. Innerhalb des Vertrags verständigen sich die Partner auf die Bedeutung der definierten Funktionalitäten. Aus diesem Grund existieren bei der Erstellung des formalen und ausführbaren Prozessmodells (z.B. mittels BPEL) für den Entwickler der Fachseite keine Unklarheiten über die Bedeutungen der Operationen und Datentypen aus der WSDL-Beschreibung. In einem „contract last“-Entwicklungsszenario wird der Dienst vor der Definition des Vertrags implementiert. Die Dienstbeschreibung wird aus der Implementierung generiert. Wird die Beschreibung durch die Dienstkonsumenten akzeptiert, kann der Vertrag als geschlossen betrachtet werden. Ein Austausch über die Bedeutungen der Operationen und Datentypen sollte jedoch auch hier vor Vertragsabschluss stattfinden; d.h. sobald der Entwickler eines Prozessmodells eine Dienstbeschreibung akzeptiert, sollte er die Beschreibung auch korrekt interpretieren können.

Mit der akzeptierten Dienstbeschreibung wird von der Fachseite ein ausführbares Prozessmodell entwickelt. Während der Erstellung des Modells werden einzelne Aktivitäten auf die Funktionalitäten der Dienste abgebildet. Falsche Abbildungen sollten dadurch ausgeschlossen sein, dass die Fachseite den Vertrag mit dem Dienst akzeptiert und somit die Bedeutung der Operationen bzw. Datentypen verstanden hat. Die Prozessengine koordiniert gemäß dem Prozessmodell die Aufrufe an den statischen Diensten. In Abbildung 16 werden diese Zusammenhänge visualisiert.

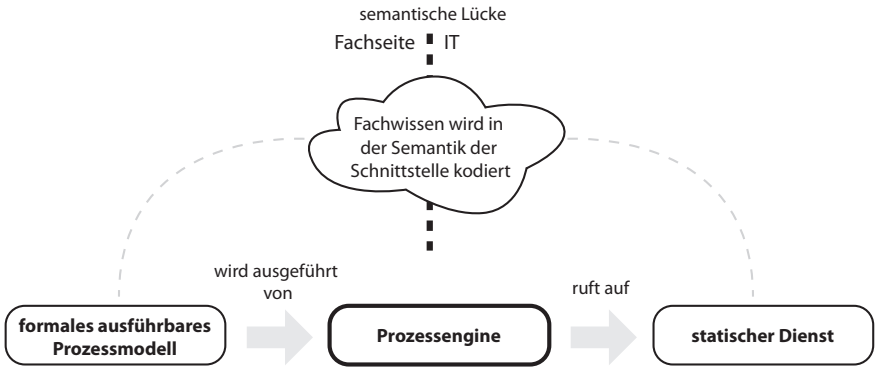

Abbildung 16: Bei Diensten nach den WS*-Technologien wird die semantische Lücke während der Beschreibung der Schnittstelle überbrückt.

6.2.2 Auswirkungen des REST-Stil

Da bei REST-Architekturen keine maschinenlesbare Dienstbeschreibung und damit auch kein expliziter Vertrag zwischen Dienstkonsumenten und -anbietern existiert, muss die Überbrückung der semantische Lücke nicht bei der Definition einer Schnittstellenbeschreibung erfolgen. Betrachtet man die Kompositionsansätze „BPEL for REST“ [33] bzw. JOpera [34] von Pautasso, so wird deutlich, dass die semantische Lücke während der Definition des Prozessmodells geschlossen wird. Am Beispiel von JOpera wird deutlich, auf welche Aspekte eines Prozessmodells die Fachseite bzw. die IT-Seite stärkeren Einfluss nehmen.

Im Kontrollflussmodell spiegelt sich implizit die Semantik der Aktivitäten des Geschäftsprozesses wider, da deren Abfolge von der Bedeutung der jeweiligen Aktivität abhängt. Das Datenflussmodell ist dem Kontrollfluss untergeordnet. Zu seiner Erstellung ist es notwendig, dass die Bedeutungen der Ressourcen und ihrer Verknüpfungen verstanden werden. Diese Bedeutungen resultieren aus der Implementierung der Dienste. Während der Transformation des Kontrollflussmodells in ein Datenflussmodell muss die Frage geklärt werden, welche Funktionalität welches Dienstes dazu geeignet ist, die Bedeutung einer Aktivität des Kontrollflussmodells korrekt umzusetzen. Dieses Problem wird in den vorgestellten Arbeiten manuell z.B. durch den Dialog zwischen Fach- und IT-Seite gelöst. Abbildung 17 zeigt die Verlagerung der Überbrückung der semantischen Lücke.

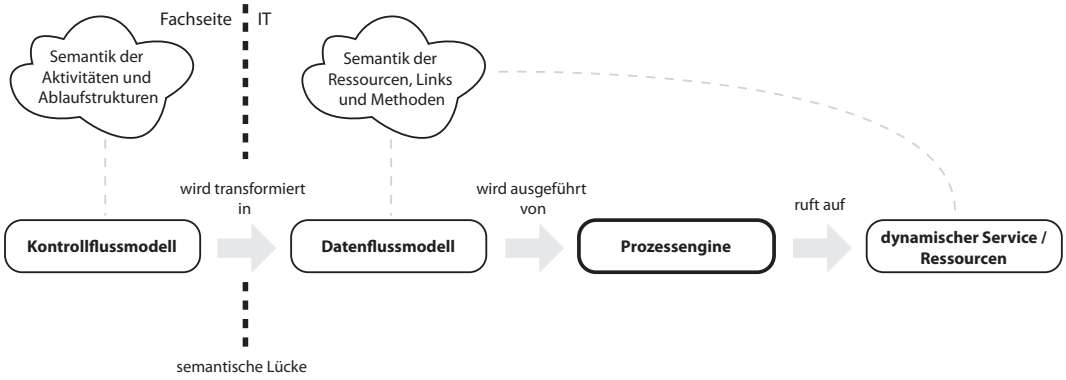

Abbildung 17: Bei RESTful Services muss die semantische Lücke während der Entwicklung der Prozessmodelle überbrückt werden.

Tilkov betrachtet Service Discovery ausschließlich unter dem Gesichtspunkt der Entdeckung von URIs. Unter dieser Perspektive hat er Recht, dass durch Hypermedia der Gesichtspunkt der Service Discovery in RESTful Architekturen automatisch gelöst wird. Eine Ressource kann auf „fremde“ Ressourcen verweisen und auf diese Weise die Entdeckung *zusätzlicher* Dienste durch Hypermedia transparent implementieren. Betrachtet man jedoch die Komposition von RESTful Services[24], so fällt auf, dass hier eine neue Art der Entdeckung notwendig wird. Es stellt sich die Frage, welchen Links gefolgt werden muss, um zu einer bestimmten Ressource zu gelangen und dort dann eine Zustandsänderung gemäß der Semantik einer Aktivität zu erreichen.

6.2.3 Semantische Lücke als unlösbares Problem

Das Problem der semantischen Lücke wird immer bestehen, da aus einer hohen, abstrakten Sprache, in diesem Fall der natürlichen Sprache, in eine formale Sprache, in diesem Fall eine Prozessmodellierungssprache abgebildet werden muss, die ggf. eine formalen Semantik besitzt. Dies lässt sich mit der Churchschen These auf einer theoretischen Basis begründen. Da die Churchsche These bisher noch nicht widerlegt werden konnte, kann sie als korrekt angesehen werden. Sie sagt aus, dass alles Berechenbare schon von einer Turing-Maschine berechnet werden kann (siehe [11]). Sollte es also Dinge geben, die nicht von einer Turing-Maschine berechnet werden können, so ist es ein nicht berechnbares Problem. Der intuitive Begriff des „Berechenbaren“ ist identisch zum Turing-Maschinen-Berechenbaren. Dabei handelt es sich um eine unbeweisbare These, da intuitive Begriffe keine mathematische Grundlage besitzen (siehe [11]).

Die Menge der formalen Sprachen ist genau so groß wie die Menge der natürlichen Zahlen (siehe [11]). Es existieren jedoch auch weitere Sprachen, zu denen keine charakteristische Funktion ermittelbar ist. Bei dieser Ermittlung handelt es sich um unentscheidbare

[24] Tilkov distanziert sich selbst von der Notwendigkeit für die Orchestrierung von Diensten (siehe [45] S. 190-191)

Probleme. Ursache hierfür sind z.B. Mehrdeutigkeiten wie in natürlicher Sprache. Solche in entscheidbare Probleme reduziert werden, wenn das ursprüngliche Problem durch zusätzliche Annahmen eingeschränkt wird.

Daraus leitet sich ab, dass die Abbildung von Prozessaktivitäten auf Dienstfunktionalitäten ohne die Hinzunahme von Annahmen immer ein Problem sein wird, das sich nicht alleine durch einen Computer lösen lässt. Somit stellen auch semantische Dienstbeschreibungen keine Lösung für das Problem der Service Discovery dar. Durch semantische Dienstbeschreibungen wird das Problem in eine höhere Abstraktionsschicht verschoben.

Aus einer pragmatischen Perspektive lässt sich dies, wie folgt, verdeutlichen: Ein Entwickler der Fachseite modelliert einen Prozess, indem er mittels einer formalen Modellierungssprache (z.B. BPEL oder BPMN) Aktivitäten erzeugt und mit einer natürlichsprachigen Beschreibung versieht. Außerdem sei vorausgesetzt, dass ein Dienst formal semantisch beschrieben ist. Nun stellt sich die Frage, wonach ein Computer entscheiden soll, welche Aktivitätsbeschreibung für welcher Dienstfunktionalität passt. Es gibt keine Möglichkeit für den Computer diesen Zusammenhang ohne zusätzliche Annahmen herzustellen, da natürliche Sprache keine formale Sprache ist.

Setzt man jedoch voraus, dass die Aktivitäten statt einer natürlichsprachigen Beschreibung eine formal semantische Beschreibung besitzen, so wäre eine automatisierte Diensterkennung möglich. Bei der Zuweisung von formaler Semantik zu den einzelnen Aktivitäten handelt es sich um einen manuell auszuführenden Schritt. Ob dieser praktikabel ist, hängt vom Willen der Entwickler, der verwendeten Entwicklungsmethodik etc. ab.

7 Fazit

7.1 Zusammenfassung der Problemstellung

Aktuelle Entwicklungen im Umfeld der verteilten Anwendungen zeigen, dass REST bzw. RESTful HTTP im öffentlichen Netz zunehmend an Bedeutung gewinnt und bereits existierende SOAP-Schnittstellen aufgegeben werden. In den SOAs von Unternehmen wird heute zu großen Teilen mit WS*-Technologien gearbeitet. Daher müssen sich Unternehmen mit der Problematik auseinandersetzen, wie sie mit REST-Architekturen umgehen, wenn sie externe RESTful Services nutzen müssen oder eigene RESTful Services anbieten wollen.

Der bewusste Verzicht auf statische Verträge zwischen Dienst und Konsument ist ein Aspekt, der vor allem bei der Entwicklung von Clients ein Umdenken erfordert. Ziel ist es dabei, eine freiere Evolution für die Dienste zu ermöglichen. Konzeptionelle Unterstützung für diese Bestrebung wird durch Hypermedia übernommen, da durch Verknüpfungen die Ablaufabhängigkeiten und Adressen der verwendeten Ressourcen zum Dienstkonsumenten kommuniziert werden. Für Dienstanbieter, Dienstkonsumenten und Kompositionswerkzeuge führt dies zur Anforderung das HATEOAS-Prinzip umzusetzen.

Der Wegfall einer formalen Dienstbeschriebung verändert die Art und Weise wie Fach- und IT-Seite die Semantik eines Dienstes definieren. Entwickler eines Clients können nicht davon ausgehen, dass sämtliche Funktionalitäten eines Dienstes zur Entwicklungszeit bekannt sind. Daher müssen Integrationsansätze entwickelt werden, um eine Etablierung von REST neben den bereits verwendeten Technologien in Unternehmen zu erreichen.

Dadurch, dass REST lediglich ein Architekturstil und kein Standard ist, muss zur Entwicklung von REST-konformen Anwendungen auf Frameworks zurückgegriffen werden. Zwar existiert bereits eine Vielzahl solcher Rahmenwerke, jedoch zeigt sich, dass nicht immer REST-Prinzipien beachtet werden

7.2 Methodik

Zu Beginn der Arbeit ist eine Einführung in die Prinzipien des Architekturstils REST gegeben worden. Durch die Definition weiterer Begriffe wie der *losen Kopplung* und *Interoperabilität* sind erstrebenswerte generische Anforderungen an Anwendungen beschrieben worden, die vor allem in einer SOA eine große Bedeutung besitzen. Hieran angeschlossen, sind bisher in SOAs eingesetzte Konzepte und Technologien erläutert worden. Zur Festigung des Verständnis sind auf ein Fallbeispiel die Konzepte praktisch angewandt worden.

Durch die Transformation des Fallbeipspiels in unterschiedliche Formen sind die Funktionalitäten identifiziert worden, die von den einzelnen getrennten Diensten der SOA des Fallbeispiels geleistet werden müssen. Diese Dienste und die zugehörigen Clients sind exemplarisch implementiert, so dass die REST-Prinzipien auf eine RESTful HTTP Implementierung abgebildet wurden. Dadurch lässt sich die Konformität der existierender Frameworks zu REST beurteilen. In einer Negativ-Implementierung ist bewusst gegen REST-

Prinzipien verstoßen worden, um auf häufig gemachte Implementierungsfehler hinzuweisen.

Ausgehend von der Implementierung ist der Umgang mit Hypermedia detailiert betrachtet worden, da er für REST ein wesentliches Unterscheidungsmerkmal zu anderen Archiekturstilen ist. Teil dieser Betrachtung sind das HTTP, die JAX-RS-Spezifikation und mehrere Frameworks. Als Ergebnis geht hierbei eine Beurteilung der Client-Stile und der „REST-fulness“ der Frameworks hervor. Außerdem wird dabei deutlich, dass die Verwendung des HTTP allein nicht automatisch zur einer REST-konformen Anwendung führt.

Mit dem Wissen, dass das HATEOAS-Printip ein wesentliches Unterscheidungsmerkmal zu anderen Architekturstilen darstellt, kann die Thematik der Komposition von REST-ful Services betrachtet werden. Dazu sind mehrere Ansätze kritisch untersucht worden, was sie zu einer Dienstkomposition beitragen können, ohne dabei gegen die REST-Prinzipien zu verstoßen. Die einzelnen Ansätze sind anschließend anhand der Probleme aus der Einleitung dieser Arbeit beurteilt worden.

Anschließend ist mittels der Anwendung eines Kompositionsansatzes gezeigt worden, dass die Integration von RESTful und WS*-Services innerhalb eines Kompositionsmodells möglich ist. Diese Hybridität ist für realistische Szenarien von zentraler Bedeutung, jedoch wird diese in den wissenschaftlichen Arbeiten nicht bzw. nur wenig thematisiert. Abschließend ist eine Empfehlung darüber abgegeben worden, wann welcher Ansatz am sinnvollsten eingesetzt werden kann.

Durch Aufzeigen eines Spannungsverhältnis zwischen Dienstdynamik und Technologiebefreiung der Prozessmodelle wird verdeutlicht, dass keine der vorgestellten Kompositionsmethodik zu einer idealen Architektur führt, sondern dass es bei der Verwendung von REST immer um individuelle Kompromisse geht. Der Gedanke der idealen Architektur, die maximale Dynamik der Dienste und maximale Technologiebefreiung bei der Prozessmodellierung bietet, ist dazu verwendet worden, um das anliegende Themenfeld „Service Discovery“ aufzuzeigen.

Der Begriff der „semantischen Lücke“ ist definiert und auf das SOA-Szenario angewandt worden, um zeigen, dass diese eine Ursache für die Notwendigkeit von Service Discovery darstellt. Mittels der Churchschen These aus dem Fachbereich der theoretischen Informatik ist verdeutlicht worden, dass keine Architektur und kein Architekturstil die Kommunikation zwischen IT- und Fachseite bei Entwicklung von Prozessmodellen überflüssig macht.

7.3 Arbeitsergebnis

Diese Arbeit hat gezeigt, dass der durch REST eingeführte Verzicht auf eine Schnittstellenbeschreibung kein unüberwindbares Problem für die Implementierung von Dienstkonsumenten darstellt. Zwar existiert mit WADL eine auf das HTTP zugeschnittene Beschreibungssprache, jedoch ist diese nicht nach den REST-Prinzipien ausgerichtet. Durch die Verwendung von WADL wird es Entwicklern ermöglicht, gegen die Regeln einer Anwen-

dungsdomäne zu verstoßen, da der Dienst keinen Einfluss auf die Steuerung des Anwendungszustands besitzt. Mit ReLL ist eine Beschreibungssprache vorgestellt worden, die an dieser Stelle ansetzt und Hypermedia explizit mit in die Schnittstellenbeschreibung einbezieht. Die Komposition von RESTful Services ist jedoch nicht zwingend auf eine zusätzliche Beschreibung der Dienste angewiesen. JOpera und „BPEL for REST“ zeigen, dass Informationen über eine Dienstschnittstelle direkt in den Kompositionsmodellen festgelegt werden können. Die von Dienstkonsumenten erwarteten Verknüpfungen der Ressourcen sind in diesem Fall direkt innerhalb der Modelle kodiert.

Da die Semantik eines Dienstes nicht mehr während der Definition eines Vertrags zwischen Dienstanbieter und Konsument festgelegt wird, kann ein Dienstkonsument nicht mehr auf eine statische Menge von semantisch bekannten Funktionalitäten zurückgreifen. Stattdessen wird den Diensten durch REST eine größere Dynamik eingeräumt, die durch die Dienstkonsumenten beachtet werden muss. Um den Entwicklern der Clients die Semantik eines RESTful Services zugänglich zu machen, müssen die Zusammenhänge zwischen Ressourcen und Bedeutungen von Ressourcen verständlich gemacht werden. Hier stellt ReLL einen möglichen Ansatz dar, der bewusst nicht versucht, zu bereits existierenden Sprachen kompatibel zu sein. Stattdessen wird der Fokus ausschließlich auf die REST-Prinzipien gerichtet. Außerdem bietet das uniforme Interface von REST den Vorteil, dass alle Funktionalitäten durch einheitlich ansprechbare Ressourcen umgesetzt werden. Die fehlende domänenspezifische Semantik verlagert sich von Operationen hin zu miteinander verknüpften Ressourcen. Durch die Verknüpfungen werden Bedeutungszusammenhänge explizit in die Schnittstelle der Dienste aufgenommen.

Für den Entwickler eines Clients bedeutet dies, dass er bei der Implementierung seine gewünschte Funktionalität auf verknüpfte Ressourcen und die Methoden des uniformen Interface abbilden muss. Damit die Überbrückung der semantischen Lücke korrekt durchgeführt wird muss, ein Dialog zwischen den Client-Entwicklern und den Dienstanbietern erfolgen. Dieser Dialog ist keine Neuerscheinung, die durch REST hinzugekommen ist, sondern er existierte bereits bei Definition eines statischen Schnittstellenvertrags. Unabhängig von einem konkreten Architekturstil wird dieser Dialog auch in Zukunft erforderlich sein.

An Hand von JOpera lässt sich aufzeigen, dass die Bedeutungen eines Dienstes bzw. der Anwendungsdomäne während des Übergangs von Ablaufabhängigkeiten in den Datenfluss geklärt werden muss.

Hypermedia trägt maßgebend zum Verständnis der Dienstsemantik bei, da hierdurch Zusammenhänge zwischen Ressourcen beschrieben sowie die möglichen Zustandsänderungen der Ressourcen gesteuert werden. Daher ist es notwendig, HATEOAS nicht als optionales Prinzip von REST zu betrachten, sondern alle Anwendungen innerhalb einer REST-Architektur nach der Verwendung von Hypermediainformationen auszurichten. Diese Arbeit zeigt, dass in der JAX-RS-Spezifikation und einigen Frameworks Nachbesserungen erforderlich sind und im Fall von JAX-RS sogar bereits in der Entwicklung sind, dies in

die Realität umzusetzen. Allein die Verwendung des HTTP zur Kommunikation zwischen Systemkomponenten führt nicht automatisch zu REST-konformen Anwendungen. als Aktuell bietet Restfulie Anwendungsentwicklern die meiste Unterstützung beim Umgang mit Hypermediainformationen. Die Jersey-Erweiterung von Hadley ist als ein exploratives Experiment entstanden und somit nicht für den Produktiveinsatz geeignet. Spring Web MVC und RESTeasy überlassen den Umgang mit Hypermediainformationen vollständig dem Anwendungsentwickler.

Die vorliegende Arbeit zeigt außerdem, dass die Integration SOAP und RESTful HTTP Diensten in Kompositionen möglich ist. Hierbei sind Entwickler vor die Wahl gestellt, BPEL und zugehörige Ausführungsumgebungen zu erweitern oder auf alternative Technologien wie z.B. JOpera zu migrieren. JOpera bietet den Vorteil, dass es nicht auf ein einziges Protokoll festgelegt ist, wie es bei BPEL der Fall ist. Ermöglicht wird dies durch die Trennung des Ablaufs eines Prozesses von den Interaktionen mit Diensten. Zwar gelingt JOpera die Trennung nicht exakt, da die verwendeten Technologien Auswirkungen auf die Ablaufabhängigkeiten des Prozesses haben können. Jedoch bietet der Ansatz Potenzial für Verbesserungen.

7.4 Ausblick

Der aktuelle Trend zeigt, dass sich REST-Architekturen weiter verbreiten. Hierfür existieren mehrere Gründe: Zum einen kann eine RESTful HTTP-Schnittstelle sowohl von Menschen als auch von Maschinen genutzt werden. In Anhang A.2 ist dies anhand von in JOpera modellierten Prozessen gezeigt. Zum anderen sind RESTful HTTP Dienste direkt aus einem Browser heraus nutzbar, da sie keine zwischengeschaltete Anwendung für die Erstellung einer Browser geeigneten Darstellung benötigen. Sobald das HATEOAS-Prinzip weitere Verbreitung in den Frameworks und Anwendungen findet, wird dieses vermutlich ebenso zu einem vermehrten Einsatz von REST-Architekturen beitragen. Grund hierfür sind der größere Evolutionsfreiraum für Dienste und die Möglichkeit über die Dienstschnittstelle Geschäftsregeln zu exponieren.

Da REST nur ein Architekturstil und keine Implementierung darstellt, werden sich zukünftige Entwicklungen stärker auf die Umsetzung von konkreten Architekturen und Technologien richten, die den REST-Prinzipien folgen. Eine Möglichkeit zur Verbesserung besteht seitens des Kompositionswerkzeugs JOpera. Hier ist eine strengere Trennung der Ablaufabhängigkeiten vom Datenfluss erstrebenswert, um Kompositionsmodelle für RESTful Services mit weniger technologischen Details zu belasten. Zwei Ansatzpunkte sind hierfür denkbar. Zum einen ließen sich durch die Bindung mehrerer Adapter an eine Aktivität auf Grund der verwendeten Technologie benötigte Aktivität vermeiden bzw. reduzieren. Zum anderen könnte die Verbindung von JOpera mit einer Ressourcenbeschreibungssprache wie ReLL dazu beitragen, Details zur Extraktion von URIs aus den Kompositionsmodellen auszuschließen.

Hinsichtlich der Vereinheitlichung und Vereinfachung der Client-Programmierung bleibt

abzuwarten, was die kommende JAX-RS 2.0-Spezifikation hierzu beitragen kann. Das Ziel der besseren Unterstützung des HATEOAS-Prinzips ist jedoch richtig gewählt.

Aus der Perspektive des Entwicklungsprozesses sind weitere Forschungsarbeiten denkbar, die sich mit dem Thema der Service Discovery für RESTful Services beschäftigen. Ziel sollte dabei die Suche nach geeigneten Annahmen sein, um das Problem der semantischen Lücke zu reduzieren und anschließend automatisiert lösen zu lassen. Ein möglicher Ansatz hierfür könnte die Verwendung von formal semantischen Ressourcenbeschreibungen und Kompositionsmodellen darstellen.

Abschließend ist festzuhalten, dass in der Geschichte keine Technologien ewig verwendet wurde. Dies wird vermutlich auch für REST gelten. Im Moment ist jedoch genug Potential für Innovationen um diesen Architekturstil herum vorhanden und viele Webentwickler fühlen sich vom WS*-Technologie-Stack nicht unterstützt. Daher wird REST in den nächsten Jahren weiterhin ein breites Forschungsumfeld bieten.

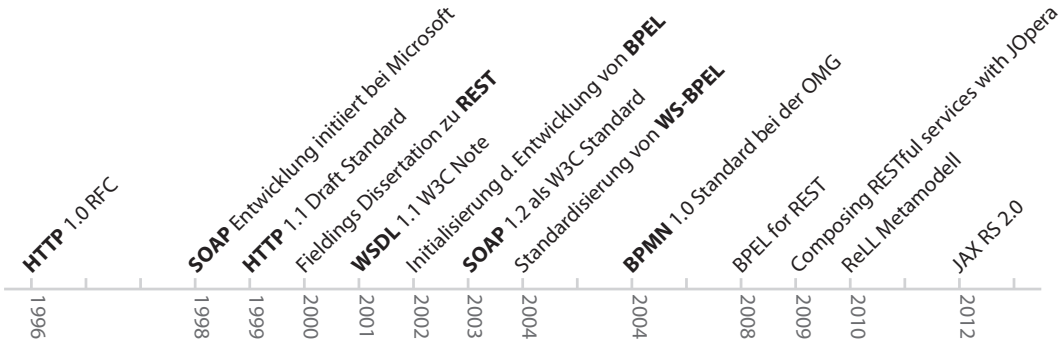

Abbildung 18: Historie der Entwicklung von Webtechnologien

Literatur

[1] Wil Van Der Aalst, Kees Van Hee und Wil Van Der Aalst. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. Mit Press, 2004.

[2] Rosa Alarcon und Erik. Wilde. From RESTful Services to RDF: Connecting the Web and the Semantic Web. *UC Berkeley: School of Information. Report 2010-041.*, 2010.

[3] Rosa Alarcon und Erik Wilde. Linking Data from RESTful Services. *Proceedings of the WWW2010 Workshop on Linked Data on the Web*, 2010.

[4] Rosa Alarcon, Erik Wilde und Jesus Bellido. Hypermedia-Driven RESTful Service Composition. In: E. Maximilien, Gustavo Rossi, Soe-Tsyr Yuan, Heiko Ludwig und Marcelo Fantinato: *Service-Oriented Computing*, Band 6568 von *Lecture Notes in Computer Science*, Seiten 111–120. Springer Berlin / Heidelberg, 2011.

[5] Thomas Allweyer. *BPMN 2.0 Business Process Model and Notation, Einführung in den Standard für die Geschäftsprozessmodellierung*. Books on Demand GmbH, Norderstedt, 2. Auflage, 2009.

[6] T. Berners-Lee, R. Fielding und H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. 1996. http://www.ietf.org/rfc/rfc1945.txt, Abruf: 2.6.2011.

[7] Erik Christensen, Francisco Curbera, Greg Meredith und Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. 2011. http://www.w3.org/TR/wsdl.html, Abruf: 29.1.2011.

[8] Andrzej Cichocki, Helal A. Ansari, Marek Rusinkiewicz und Darrell Woelk. *Workflow and process automation: concepts and technology*. Springer-Verlag, 1998.

[9] Gero Decker, Alexander Lüders, Hagen Overdick, Kai Schlichting und Mathias Weske. RESTful Petri Net Execution. In: Roberto Bruni und Karsten Wolf: *Web Services and Formal Methods*, Band 5387 von *Lecture Notes in Computer Science*, Seiten 73–87. Springer Berlin / Heidelberg, 2009.

[10] Brian Elvesäeter, Dima Panfilenko, Sven Jacobi und Christian Hahn. Aligning business and IT models in service-oriented architectures using BPMN and SoaML. In: *Proceedings of the First International Workshop on Model-Driven Interoperability*, MDI '10, Seiten 61–68, New York, NY, USA, 2010. ACM.

[11] Katrin Erk und Lutz Priese. *Theoretische Informatik - Eine umfassende Einführung*. Springer Berlin / Heidelberg, 3. Auflage, 2008.

[12] Alberto Fernandez, Conor Hayes, Nikos Loutas, Vassilios Peristeras, Axel Polleres und Konstantinos A. Tarabanis. Closing the Service Discovery Gap by Collaborative Tagging and Clustering Techniques. In: Rubén Lara Hernandez, Tommaso Di Noia

und Ioan Toma: *SMRR*, Band 416 von *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. 1999. http://www.ietf.org/rfc/rfc2616.txt, Abruf: 8.5.2011.

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[15] Otávio Freitas Ferreira Filho, Maria Alice Grigas und Varella Ferreira. Semantic Web Services: A RESTful Approach. *Proceedings of IADIS International Conference WWW/Internet 2009 Rome*, Seiten 169–180, 2009.

[16] Marc Hadley, Santiago Pericas-Geertsen und Paul Sandoz. Exploring hypermedia support in Jersey. In: *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, Seiten 10–14, New York, NY, USA, 2010. ACM.

[17] Marc Hadley und Paul Sandoz. JAX-RS: Java API for RESTful Web Services. 2008. http://jsr311.java.net/nonav/releases/1.0/spec/index.html, Abruf: 29.3.2011.

[18] Mark D. Hansen. *SOA Using Java Web Services*. Prentice Hall, 2007.

[19] Anthony Hock-koon und Mourad Oussalah. Specifying Loose Coupling from Existing Service Composition Approaches. In: Muhammad Babar und Ian Gorton: *Software Architecture*, Band 6285 von *Lecture Notes in Computer Science*, Seiten 464–471. Springer Berlin / Heidelberg, 2010.

[20] Mike Kelly und Michael Hausenblas. Using HTTP link: header for gateway cache invalidation. In: *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, Seiten 23–26, New York, NY, USA, 2010. ACM.

[21] Ann Lindsay, Denise Downs und Ken Lunn. Business processes–attempts to find a definition. *Information and Software Technology*, 45(15):1015 – 1019, 2003. Special Issue on Modelling Organisational Processes.

[22] Dieter Masak. Geschäftsprozess. In: *SOA?*, Xpert.press, Seiten 175–190. Springer Berlin Heidelberg, 2007.

[23] Kunal Mittal. Build your SOA, Part 1: Maturity and methodology. 2005. http://www.ibm.com/developerworks/webservices/library/ws-soa-method1.html, Abruf: 23.3.2011.

[24] M. Mongiello und D. Castelluccia. Modelling and verification of BPEL business processes. In: *Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006. Fourth and Third International Workshop on*, march 2006.

[25] Chun Ouyang, Marlon Dumas und Arthur H. M. Ter Hofstede. Pattern-based Translation of BPMN Process Models to BPEL Web Services.

[26] o.V. JOpera Manual. 2010. http://www.jopera.org/docs/help/jop_1.html, Abruf: 23.6.2011.

[27] o.V. JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services. 2011. http://jcp.org/en/jsr/proposalDetails?id=339, Abruf: 3.6.2011.

[28] o.V. Apache CXF. o.J. http://de.wikipedia.org/wiki/Apache_CXF, Abruf: 6.5.2011.

[29] o.V. RESTFul Web Services for Java. o.J. http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html_single/index.html, Abruf: 11.5.2011.

[30] o.V. WS-BPEL FAQ. o.J. http://www.oasis-open.org/committees/download.php/23858/WS-BPEL-2.0-FAQ.html, Abruf: 29.1.2011.

[31] Hagen Overdick. The Resource-Oriented Architecture. In: *IEEE SCW*, Seiten 340–347. IEEE Computer Society, 2007.

[32] Emilian Pascalau und Adrian Giurca. A Rule-Based Approach of Creating and Executing Mashups. In: Claude Godart, Norbert Gronau, Sushil Sharma und Gérôme Canals: *Software Services for e-Business and e-Society*, Band 305 von *IFIP Advances in Information and Communication Technology*, Seiten 82–95. Springer Boston, 2009.

[33] Cesare Pautasso. *BPEL for REST*. Springer-Verlag Berlin Heidelberg, 2008.

[34] Cesare Pautasso. Composing RESTful Services with JOpera. In: Alexandre Bergel und Johan Fabry: *Software Composition*, Band 5634 von *Lecture Notes in Computer Science*, Seiten 142–159. Springer Berlin / Heidelberg, 2009.

[35] Cesare Pautasso, Olaf Zimmermann und Frank Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. *WWW '08 Proceeding of the 17th international conference on World Wide Web*, April 2008.

[36] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oktober 2003.

[37] Kerstin Pfitzner, Gero Decker, Oliver Kopp und Frank Leymann. Service-Oriented Computing - ICSOC 2007 Workshops. chapter Web Service Choreography Configurations for BPMN, Seiten 401–412. Springer-Verlag, Berlin, Heidelberg, 2009.

[38] Arjen Poutsma. REST in Spring 3: @MVC. 2009. http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc/, Abruf: 12.6.2011.

[39] Arjen Poutsma. REST in Spring 3: RestTemplate. 2009. http://blog.springsource.com/2009/03/27/rest-in-spring-3-resttemplate, Abruf: 10.6.2011.

[40] Tatu Saloranta. Using JAXB annotations with Jackson. Januar 2011. http://wiki.fasterxml.com/JacksonJAXBAnnotations, Abruf: 8.5.2011.

[41] Andreas Schmidt. Integrationsansätze: Überblick und Potentiale semantischer Technologien. 2007. http://www.slideshare.net/aps/integrationsanstze-berblick-und-potentiale-semantischer-technologien, Abruf: 20.6.2011.

[42] L. Simon, Ajay Mallya, Ajay Bansal, Gopal Gupta und T.D. Hite. A Universal Service Description Language. In: *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, Seiten 2 vol. (xxxiii+856), Juli 2005.

[43] A.W.M. Smeulders, M. Worring, S. Santini, A. Gupta und R. Jain. Content-based image retrieval at the end of the early years. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(12):1349 –1380, Dezember 2000.

[44] Stefan Tilkov. JSR 311 Final: Java API for RESTful Web Services. 2008. http://www.infoq.com/news/2008/09/jsr311-approved, Abruf: 3.6.2011.

[45] Stefan Tilkov. *REST und HTTP Einsatz der Architektur des Web für Integrationsszenarien*. dpunkt Verlag, Juli 2009.

[46] K. Vergidis, A. Tiwari und B. Majeed. Business Process Analysis and Optimization: Beyond Reengineering. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(1):69–82, Januar 2008.

[47] Xiwei Xu, Liming Zhu, Yan Liu und Mark Staples. Resource-Oriented Architecture for Business Processes. *APSEC '08 Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, 2008.

[48] Xiwei Xu, Liming Zhu, Yan Liu und Mark Staples. Resource-Oriented Business Process Modeling for Ultra-Large-Scale Systems. *ULSSIS '08 Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, 2008.

A Anhang

A.1 Implementierungen

Sämtlicher Programmcode, der während der Entstehung dieser Arbeit geschrieben wurde, ist auf dem beigefügten Datenträger enthalten. Um den Programmcode zu kompilieren sollte Apache Maven verwendet werden. In so fern es nicht bereits installiert ist, befinden sich unter `http://maven.apache.org/index.html` Informationen dazu, wie Maven installiert und benutzt werden kann.
Es existieren drei voneinander getrennte Maven-Projekte auf dem Datenträger:

1. `maven_projects/accounting`
2. `maven_projects/ccs`
3. `maven_projects/pvp`

Über die Kommandzeile lassen sich die Projekte mittels des `build.sh`-Scripts kompilieren. Das `install.sh`-Script kopiert die kompilierten Anwendungen in das Verzeichnis des JBoss Application Server. Je nach Entwicklungsumgebung muss im Installations-Script der Pfad des Application Servers noch angepasst werden.

Statt die Projekte manuell zu übersetzen und zu installieren kann auch der JBoss Application Server aus der Datei `jboss-6.0.0.Final+deployed_appliations.zip` verwendet werden, da er die genannten Anwendungen bereits enthält. Zum Start des Application Servers muss in das `jboss-6.0.0.Final/bin`-Verzeichnis gewechselt werden und dort das `run.sh`-Script aufgerufen werden.

Nachdem der Application Server gestartet ist, was mehrere Minuten dauern kann, stehen die Anwendungen unter den folgenden Adressen bereit:

1. `maven_projects/accounting`
   → `http://localhost:8080/accounting`
2. `maven_projects/ccs`
   → `http://localhost:8080/ccs/rest`
   → `http://localhost:8080/ccs/soap`
3. `maven_projects/pvp`
   → `http://localhost:8080/pvp/contract`

Datenträger

A.2 Verwendung von JOpera

Um JOpera in Eclipse zu installieren sollte die folgende *Update Site* verwendet werden, da die aktuelle Version 2.5.4 Fehler enthält: `http://www.update.jopera.org/old/jopera2.5.3`
Das JOpera Prozessmodell aus Abschnitt 5.3 befindet sich auf dem beigefügten Datenträger im `jopera_sample`-Ordner und kann in Eclipse als JOpera-Projekt importiert werden.

Da bei der Implementierung des Fallbeispiels parallel zur JOpera Laufzeitumgebung ein JBoss Application Server betrieben wird und beide per Standard auf den Port 8080 konfiguriert sind, muss eine der beiden Systemkomponenten auf einen anderen Port umkonfiguriert werden. In der vorliegenden Implementierung wurde hierfür JOpera auf Port 8081 verlagert. Diese Einstellung ist in den Eclipse Einstellungen durchzuführen (Preferences → JOpera → Engine Threads).

Sobald das JOpera Eclipse Projekt importiert ist steht der modellierte Prozess und folgender URI zur Verwendung bereit:

`http://localhost:8081/rest/AccountingServiceService/NewProcess/1.0.`

Um eine neue Prozessinstanz zu erzeugen kann entweder das browserbasierte Frontend oder ein Skript für die Kommandozeile genutzt werden. Letzteres greift auf das Werkzeug `curl` zurück. Ein Beispielaufruf sieht wie folgt aus:

```
curl -d "Action=run" http://localhost:8081/rest/AccountingServiceService/
    NewProcess/1.0/  -d "dateofbirth=1985-01-03T11:11:28.216" -d "lastname
    =Hess" -d "bankCode=1234500" -d "country=Deutschland" -d "housenumber
    =80" -d "bankAccountNumber=758385700" -d "firstname=Thomas" -d "city=
    Koeln" -d "street=Wallstr." -d "birthcity=Leverkusen" -d "sex=MALE" -d
     "tariffFixedPrice=5.0" -d "bankName=Spasskasse" -d "
    numberMigrationRequired=false" -d "zip=51063" -d "email=mim124@gm.fh-
    koeln.de"
```

Hierdurch wird eine POST-HTTP-Anfrage mit den angegebenen Prozessparametern an JOpera gesendet, woraufhin der Prozess instanziiert und synchron bearbeitet wird.

Der Action-Parameter legt fest, ob die Durchführung des Prozesses synchron oder asynchron zur erzeugenden Anfrage erfolgt. Wenn „run“ als Wert angegeben wird, erfolgt eine synchrone Verarbeitung, während der Wert „start“ für eine asynchrone Verarbeitung verwendet werden muss.

A.3 BPMN-Modell zum Fallbeispiel

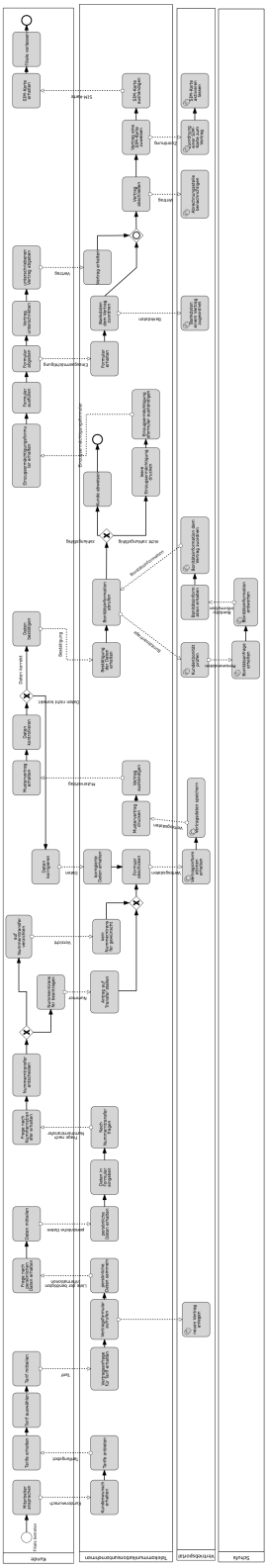

Abbildung 19: Der Geschäftsprozess des Fallbeispiels als BPMN-Modell

A.4 BPEL for REST Beispielprozess

Dieses Code-Beispiel ist aus der Arbeit „BPEL for REST“ [33] von Pautasso übernommen.

```
<process name="LoanApplication">
  <resource uri="loan">
    <!-- State variables of the resource -->
    <variable name="name"/>
    <variable name="amount"/>
    <variable name="rate"/>
    <variable name="bank"/>
    <variable name="start_date"/>
    <variable name="end_date"/>

    <!-- PUT /loan request handler -->
    <onPut>
      <if><condition>$request.amount > 100000</condition>
        <then>
          <response code="400">Requested amount too large</response>
          <exit/>
        </then>
        <else>
          <sequence>
            <assign>name = $request.name; amount = $request.amount; start_date =
                $request.start_date;</assign>
            <response code="201">Processing loan application</response>

            <!-- Get rates fromtwo different bank services -->
            <scope>
              <variable name="ubs_response" />
              <variable name="cs_response" />
              <variable name="url_accept" />
              <variable name="accept_response" />

              <flow>
                <get uri="http://www.ubs.ch/rate?chf=$amount&from=$start_date"
                    response="ubs_response" >
                <get
                    uri="http://www.cs.ch/rates?amount=$amount&start=$start_date"
                    response="cs_response" >
              </flow>

              <!-- Let client choose the preferred bank -->
              <while>
                <condition>TRUE</condition>
                <resource uri="choice" >
                  <onGet> <!-- Return the rates offered by the banks -->
                    <response code="200">
                      <header name="Content-Type">application/json</header>
                      [ {bank:"cs", rate: "$cs_response.rate" , end_date:
                          "$cs_response. until"}, {bank: "ubs", rate:
                          "$ubs_response.rate" , end_date: "$ubs_response.end"} ]
                    </response>
                  </onGet>

                  <onDelete>
                    <!-- Reject the offer and cancel the loan application -->
                    <sequence>
                      <response code="200" />
                      <exit/>
                    </sequence>
                  </onDelete>
                  <onPost>
                  <!-- Store the client choice and continue -->
                    <sequence>
                      <assign>bank = $request.choice;</assign>
                      <if><condition>bank == "cs"</condition>
                        <then>
                          <assign>rate = $cs_response.rate; end_date =
```

```
                        $cs_response.until;url_accept =
                        $cs_response.accept</assign>
                    </then>
                    <else>
                      <assign>rate = $ubs_response.rate; end_date =
                        $ubs_response. end; url_accept =
                        $ubs_response.accept </assign>
                    </else>
                  </if>
                  <response code="200" />
                  <activeBPEL:break/>
                </sequence>
              </onPost>
            </resource>
          </while>
          <!-- Accept the loan offered by the chosen bank -->
          <post uri="$url_accept" request ="$name" response="accept_response">
        </scope>
      </sequence>
    </else>
  </if>
</onPut>

<!-- GET /loan request handler -->
<onGet>
<!-- Return the state of the loan application -->
</onGet>

<!-- DELETE /loan request handler -->
<onDelete>
  <if><condition>bank == null</condition>
  <then>
    <response code="200"/>
    <exit/>
  </then>
  <else>
  <!-- Start the loan cancellation process -->
  <invoke...>
  </else>
  </if
</onDelete>
</resource>
</process>
```

A.5 Screenshots zum Fallbeispiel

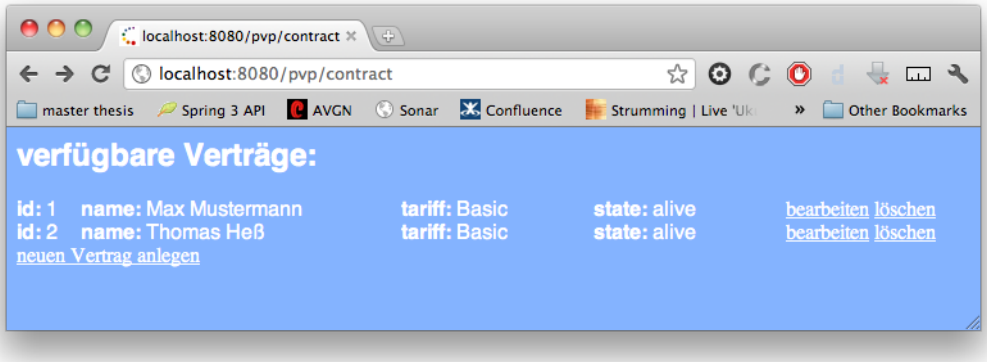

Abbildung 20: HTML-Ansicht aller Verträge im Privatkundenvertriebsportal

Abbildung 21: HTML-Formular eines Vertrags im Privatkundenvertriebsportal

Abbildung 22: Screenshot zweier mittels des cREST Client ausgeführter Anfragen an den Customer and Contract Service

Danksagung

An dieser Stelle möchte ich mich noch bei den Menschen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben:

Für sein Feedback bei der Entwicklung eines realistischen Fallbeispiels möchte ich mich bei Matthias Richter bedanken.

Für die Unterstützung gerade während der Abschlussphase geht mein Dank an meine Familie, Freundin und Freunde.

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, den 30. Juni 2011

Thomas Heß